RICE UNIVERSITY

Exploiting Generational Garbage Collection: Using Data Remnants to Improve Memory Analysis and Digital Forensics

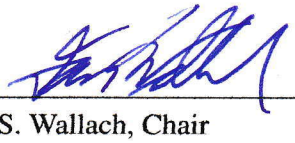
by

Adam T. Pridgen

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

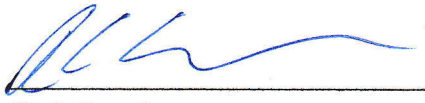
Doctor of Philosophy

APPROVED, THESIS COMMITTEE:



Dan S. Wallach, Chair
Professor of Electrical and Computer
Engineering and Computer Science

Keith D. Cooper
L. John & Ann H. Doerr Chair in Computational
Engineering, Professor, Computer Science



Chris Bronk
Assistant Professor of Department of
Information and Logistics Technology

Houston, Texas

January, 2017

ABSTRACT

Exploiting Generational Garbage Collection: Using Data Remnants to Improve Memory Analysis and Digital Forensics

by

Adam T. Pridgen

Malware authors employ sophisticated tools and infrastructure to undermine information security and steal data on a daily basis. When these attacks or infrastructure are discovered, digital forensics attempts to reconstruct the events from evidence left over on file systems, network drives, and system memory dumps. In the last several years, malware authors have been observed used the Java managed runtimes to commit criminal theft and conduct espionage.

Fortunately for forensic analysts, the most prevalent versions of Java uses generational garbage collection to help improve runtime performance. The memory system allocates memory from a managed heap. When memory is exhausted in this heap, the JVM will sweep over partitions reclaiming memory from *dead* objects. This memory is not sanitized or *zero'ed*. Hence, latent secrets and object data persist until it is overwritten. For example, sockets and open file recovery are possible even after resources are closed and purged from the OS kernel memory.

This research measures the lifetime of latent data and implements a Python framework that can be used to recover this object data. Latent secret lifetimes are experimentally measured using TLS keys in a Java application. An application is configured to be very active and minimally active. The application also utilizes raw Java sockets and Apache HTTPClient to determine whether or not a Java framework impacts latent secret lifetimes. Depending on the heap size (512MiB to 16GiB), between 10-40% of the TLS keys are recoverable from the heap, which correlates directly

to memory pressure.

This research also exploits properties to identify and recover evidence from the Java heap. The RecOOP framework helps locate all the loaded types, identify the managed Java heaps, and scan for potential objects. The framework then lifts these objects into Python where they can be analyzed further. One key findings include the fact that IO streams for processes started from within Java remained in memory, and the data in these buffers could be used to infer the program executed. Socket and data could also be recovered even when the socket structures were missing from the OS's kernel memory.

Contents

Illustrations

Tables

Chapter 1

Introduction

Digital forensics focuses on recovering artifacts and evidence for the purposes of reconstructing events. The information gathered comes from several places like; network traffic, appliances, applications logs, disk drives, memory, etc. However, forensic collection of evidence runs into challenges because the desired information is recovered after the the incident has occured.

Due to the diverse nature of information technology, effective forensics requires a copious amount of technology and software to perform analysis and interpret the results. Figure 1.1 shows an overview of tasks, activities, and high-level questions that help drive the digital forensics process. For example, if a disk drive is forensically analyzed, special hardware is required to mount devices read-only, and then software is used to read the disk at a low-level and account for each logical block and reconstruct the file system. This reconstruction must also account for discovered blocks not specified as in use because they may have been used to store illicit material. Fortunately, file systems rarely change, and their evolution is accompanied with the tools required for analysis.

Memory analysis is another area of digital forensics that relies on software and thorough analysis. Memory analysis focuses on recovering evidence from the physical memory and swap files. Software used to recover artifacts from system memory must consider the hardware architecture, operating system, memory management (namely paging), and data structure interpretation.

At large, memory analysis frameworks like Volatility and Rekall improve the ease of analyzing most physical memory dumps. Over time, these frameworks have been updated to handle common operating systems, hardware architectures, and interpretation of data structures. They also handle

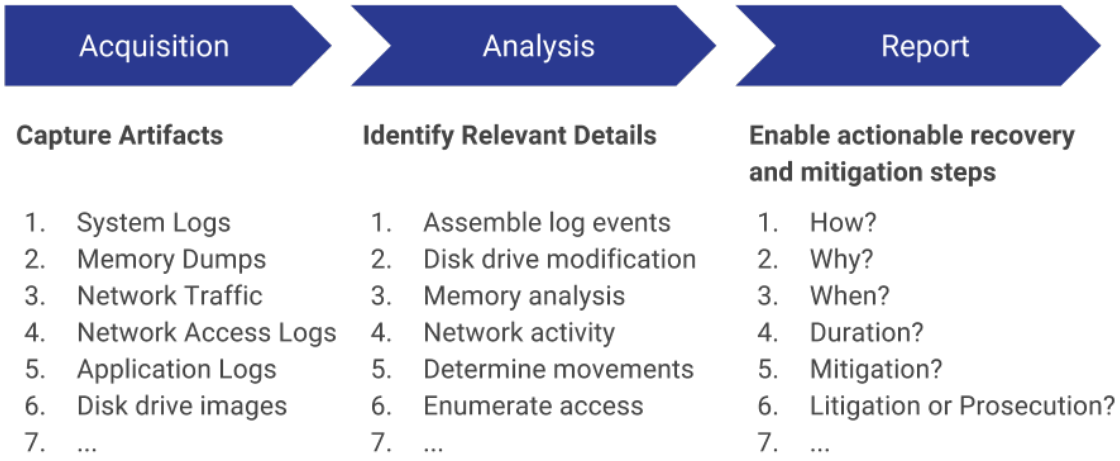


Figure 1.1 : An overview of activities and tasks that cover digital forensics.

memory images from several different formats like VMWare memory files, raw system memory, etc.

These frameworks and current memory analysis research tend to fall short when looking at applications or software systems that extend beyond the operating system. Specifically, if memory analysis is required for a specific application or process, an analyst is required to develop their own methodology and plugins on top of these frameworks.

In an enterprise environment or a law enforcement setting, this knowledge gap is daunting. As a substitution for deeper understanding, the forensic analysts use tools that look for structurally recognizable artifacts like strings, sockets, etc. This approach is used because trying to understand program data structures requires time and sometimes deep low-level understanding about how the underlying operating system and hardware work. Often, front line analysts do not have the luxury to develop the expertise and tooling to meet the demand for this specific and detailed set of knowledge.

The Stuxnet cyberattack demonstrated that almost any type equipment and software, no matter how isolated, can be used to conduct sophisticated attacks [1]. Hence, responding to sophisticated at-

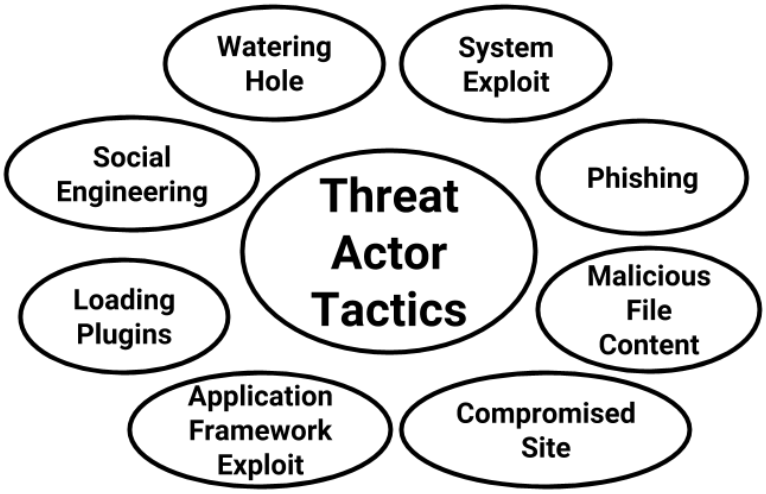


Figure 1.2 : A few attack tactics and techniques commonly employed by threat actors to gain access to victims and collateral systems.

tacks requires foresight and planning. After these revelations from Stuxnet, we surveyed common technologies in information technology to find areas that an attacker may target or use to conceal their activities. The Java virtual machine (JVM), runtime, and frameworks provide a wide spread deployment platform. Java is used in a wide array of systems ranging from Blu-Ray players to web services and applications.

In fact, threat actors have used Java for several attacks (Figure 1.2), and vulnerabilities in Java have been used to conduct these attacks and campaigns. Java malware and remote access tools (RAT) are not typically recognized as a globally prevalent threat, but implants written for the JVM still present a danger to most enterprise networks. First, this type of malware runs on almost any OS or architecture. The only requirement is compatibility between class or Java archive (JAR) files and the JVM executing the program.

Second, Java programs are typically beyond the purview of most intrusion prevention and anti-virus systems. These systems do not typically attempt to parse or understand the program. Instead they rely on hashes, regular expressions, or program behavior to identify the program and its intent.

Software virtual machines (VMs) are probably one of the most monumental inventions in information technology. The VMs on these software platforms make it possible to develop, test, and deploy code across many different hardware architectures and operating systems. The VMs abstract the native runtime away from the developer, and the VM runtime provides a rich environment for software applications. Hence, these systems have gained wide spread adoption and become the staple of modern enterprise networks. Attackers have taken note of the homogeneity of these and actively target software VMs and their respective Runtime environments. Attackers exploit the VM and the platform to gain access to the host operating system, and attackers develop malware to maintain access, which facilitates lateral movement since the runtimes on each of the machines are similar.

There are several notable campaigns against organizations using Java. Galerpin et al. detail how the Kazakh government used malicious documents to infect political dissidents and journalists [2]. In this case, the government reportedly hired a third-party security firm to spear phish individuals who were suing the Kazakh government. The attackers used a malicious PDF that installed either a Java backdoor (Jacksbot or JRat) or a Windows specific piece of malware (Bandook). JRat is a *commercially* available piece of malware with many open source modules that can be used for infecting and maintaining access on target machines.

Another campaign dubbed IceFog or JavaFog by Kaspersky labs targeted military industries, political figures, and governments [3, 4]. During the latter part of 2013, unknown threat actors targeted various organizations using Windows portable executable (PE) malware. When Kaspersky published reports detailing the IceFog campaigns, the actors shut down their infrastructure. However, a short period later, the threat actors started using a Java variant of the backdoor on US targets. At the time, the variant went undetected by several anti-virus vendors.

Scott-Railton et al. have also observed South American threat actors using Java as their primary RAT [5]. These attackers used a combination of social engineering with spear phishing, fake news sites, and malware to conduct their operations. The campaigns from these attacks were first

observed in 2015, but the activity was observed as far back as 2014. The threat actors would entice their targets into opening malicious files with convincing emails, pop-ups on malicious news sites, or using third-party cloud drives to host the malware linked from malicious sites. Between 2008-2013, the group used PE executables for remote access, but in 2014 this group moved to Java based malware, namely AlienSpy and Adzok.

The AlienSpy malware is a commercial product that has roots in an open source version from 2013. Adwind is also a variant of the same project. The criminals using Adwind have infected over 400,000 systems [6].

There are several ways to analyze the malware in live environments. Typically, this analysis requires the JAR file, an development environment like Eclipse, and in some cases, decompiled source code. The malware analyst requires the original files to make this work. For example, if the RAT downloads and deletes itself from disk, then the file may not be recoverable which prevents direct analysis. Additionally, if the malware downloads and loads modules that are on a remote system, then these analysis techniques will fail.

1.1 Supporting Work

This work builds on two of our previous projects: STAAF [7] and our Radare Java extensions [8]. STAAF is an engineering project that focused on scaling Android application analysis. In the early days of Android, off-market stores appeared online, which side-stepped some of the security protections offered in official stores supported by Google.

These off-market stores were not policed to identify potential malware or copyright infringement. As an experiment, we focused our efforts on building a scalable analyis platform. The goals of this platform were to enable collaboration among analysts, enable pluggable modules, and localize data so that modules and analysis pipelines worked efficiently. Insights learned on this project

were applied to our infrastructure for measuring latent secrets, namely orchestration, pipelining of analysis, and developing a scalable analysis system as system memories increased beyond 4GiB.

Another supporting work were our extensions to Radare. Prior to our extensions analysis of Java malware was limited to only a few tools. None of these tools permitted low-level analysis or manipulation of Java class or archive files. For instance tools like CFR and JD-GUI help with decompiling the malware. The decompiled classes are then modified and recompiled using an environment like Eclipse IDE. After recompilation, the malware can be run, monitored, and manipulated using Eclipse's integrated debuger functions. However, problems arise when the Java program cannot be decompiled.

In these cases, there were no alternatives. Our extensions helped eliminate these limitations in the following ways. First, the extensions permit the manipulation of access flags for Java classes, fields, and methods. When changed, the elements can now be accessed directly. Hence, an analyst can write a Java program or use Jython to run execute parts of the Java malware and inspect values after the computation. Furthermore, these flags can be inspected to determine whether or not they are causing faulty analysis in the decompilers or the IDE. In one example, we found that JD-GUI ignored methods with access flags marked as `native`. The `native` access flag indicates that the method is compiler generated, so our belief is JD-GUI simply ignores these functions, meaning attacker code could hide in plain sight.

Second, our extensions took advantage of Java *symbolic* references to enable hooking. Our extensions enabled us to pin-point and hook specific classes and methods. Specifically, we could read the class file, replace references to function names, and then load our own code that intercepted any calls or accesses to a target class. Renaming functions and classes in the classfile is especially challenging, because this requires resizing the entire class file.

Finally, the extensions allow an analyst to modify or directly write bytecode into methods. This injected bytecode can then be used to redirect control. Modifying instructions in the methods can

also be used to eliminate obfuscated code that is used to confuse decompilers.

1.2 Contributions

This work extends the boundaries of technical knowledge of Java forensics, malware analysis, and memory analysis in several ways. We focus exclusively on the Java runtime. Our research revealed that very little attention has been paid to the HotSpot JVM and its runtime with respect to digital forensics. We found this surprising given the maturity of the project. We demonstrate how much information can be gathered from the HotSpot JVM. Specifically, we show that object data is not sanitized or overwritten, meaning attackers or forensic specialists can easily recover *latent secrets* and other artifacts. Finally, the work demonstrates Java object recovery from a HotSpot JVM in a system memory image. Furthermore, this research generalizes how similarly managed runtimes can be analyzed for forensics purposes.

Chapter 2

HotSpot Memory Background

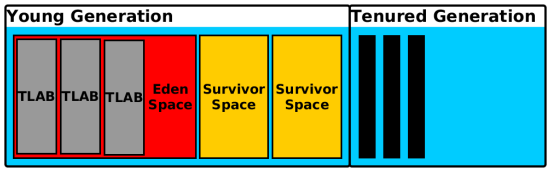
The HotSpot JVM implements several different garbage collectors, but all use *generational copying* to improve memory management performance.

In generational copying, new objects are created in the *Eden space*. The Eden space is further partitioned into *thread local allocation buffers* (TLAB), allowing for low-cost memory allocation with minimal locking in multi-threaded applications. The *generational hypothesis* states that most objects die young; indeed, most Java objects die quickly[9], but some age and survive GC, and are migrated from Eden to the *survivor spaces*, which together with the Eden Space are called the *young generation*. Objects are eventually copied from the young generation to the *tenured generation*. A typical Java heap memory layout contains many such sections (Figure 2.1a).

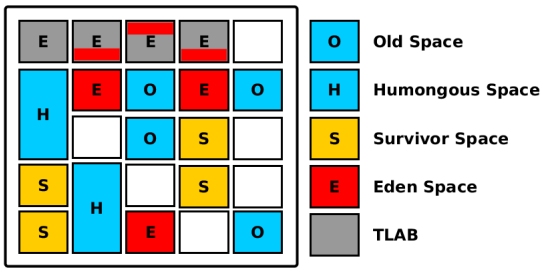
Because of its focus on performance, the JVM does not clear the contents of memory when an object is moved from one space to another [10]. Stale data will eventually be overwritten as memory is reused, but these overwrites may also never happen.

In newer HotSpot JVM's, an alternative Garbage First Garbage Collector (G1GC) uses a partitioned heap space (Figure 2.1b), allowing parallel garbage collection during incremental collection. During an incremental collection, G1GC identifies regions with the most garbage and copies the objects into a new region, allowing it to reclaim those regions [11].

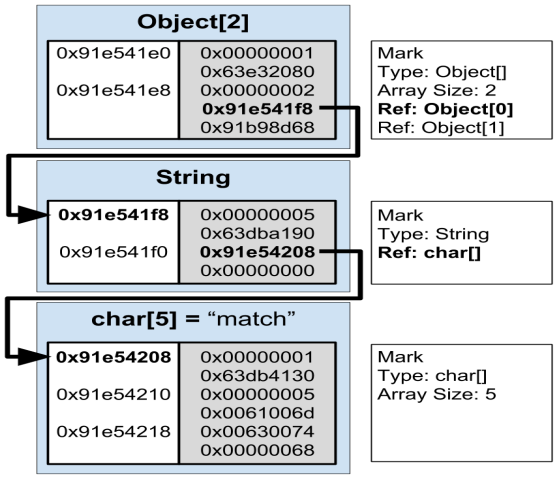
Java runtime performance typically improves when the JVM is given additional RAM, as less memory pressure results in more flexibility for the garbage collector. This decreased pressure also results in latent secrets remaining longer in RAM, improving the chances of recovering sensitive



(a) Typical SerialGC Generational Heap



(b) Typical G1GC Generational Heap



(c) OOP layout for a `String[2]` referencing `String[0]`.

Figure 2.1 : Heap and OOP memory layouts used by the HotSpot JVM.

information—a boon for forensic analysis.

Furthermore, the HotSpot JVM uses a region-based memory allocator to manage the sharing of large blocks of memory between the garbage collector and native C libraries. This creates the additional possibility that a garbage collector, finished with a region, might release it to the region allocator, which could then reuse the memory without first zeroing it.

Java objects are variably sized. The invariant part of the object structure includes a mark header, object metadata, and class information. The variable portion contains object's *non-static* fields. Raw pointers otherwise known as *original object pointers* (OOPs) refer to the address of the Java object in process memory, which lies in the heap.

The mark header usually starts the structure and includes the hash identifier of the object, thread ownership information, and metadata (e.g. age and liveness) used by the GC. This mark header is typically followed by a pointer to a type definition (which HotSpot calls a `Klass`). The type

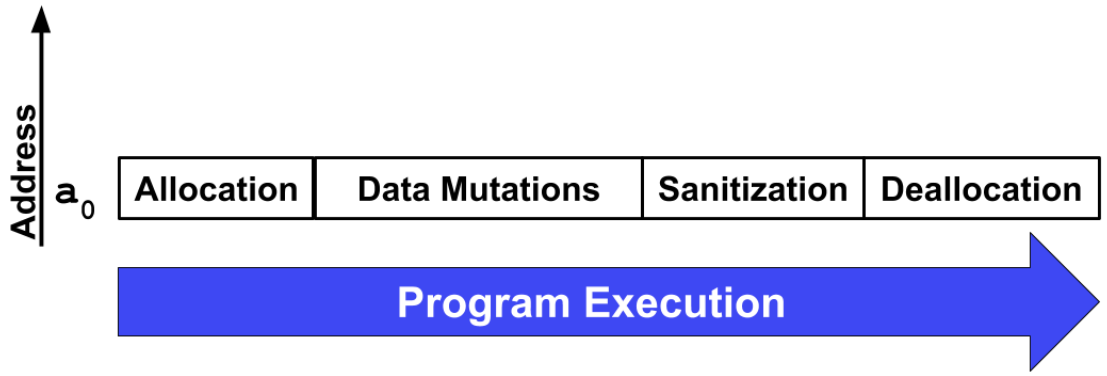
pointer defines each offset necessary to access fields of the object in the heap. If the object fields are primitive values, then these values are written directly into that memory location. If the field is a reference, then the field value is an OOP pointing to the object.

Array objects have a slightly different structure. In addition to the mark header and type information, the object also contains metadata defining dimensionality and the number of elements in the array. The size of an array object also depends on the type (e.g. `Byte[]` vs. `char[]` vs. `int[]`). The `Byte[]` is an array of OOPs, while the `char[]` and `int[]` are arrays of 2- and 4-byte values, respectively.

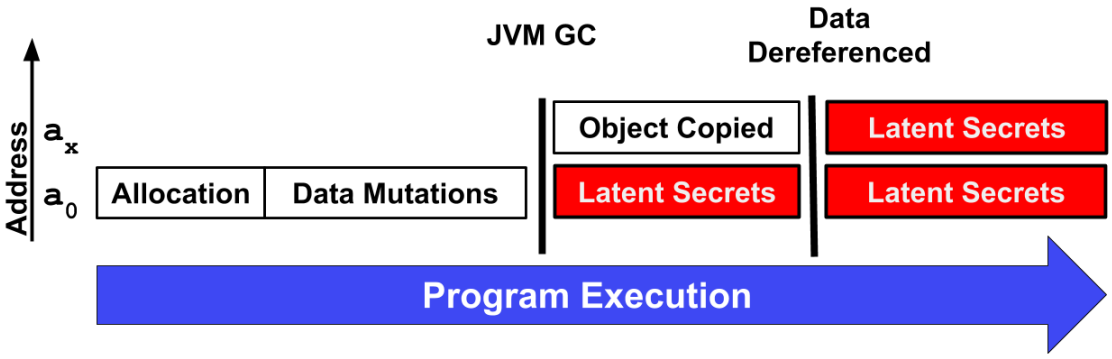
Figure 2.1c depicts the heap memory layout of a `String[2]`, which is actually an array of two `Object` references, each pointing to a `String`. The first element contains a reference to a value of type `char[5]`. The values for the `char[]` are inlined. Needless to say, these basic object header structures are kept very simple, because they will be widely repeated in memory.

Chow *et al.*defined data lifetimes from its inception to deallocation [12, 13]. These concepts are illustrated in Figure 2.2. Figure 2.2a how we expect a program dealing with sensitive data to behave. When the allocation takes place, the data structure is initialized *in-place*, and the program will perform some activity using this data. Once the data is no longer needed, the program zero's or corrupts the data and deallocates this structure. Note, the structure is not moved around once it was initialized.

This type of behavior cannot be expected in a managed memory context, especially one employing garbage collection. Figure 2.2b shows a very different story. After an object is allocated, collection can happen, and the object is copied from its original address to a new address. This creates a potential latent secret. The programmer no longer has control over the old copy. Once the data is out of scope, another latent secret is created if the developer does not corrupt the object data beforehand.



(a) Explicit data lifetime of unmanaged data.



(b) Data lifetime of GC-managed data.

Figure 2.2 : Data lifetimes in unmanged and managed vary significantly.

In runtimes using *generational* garbage collection, this problem is exacerbated. First, multiple copies of object will be created. These copies will only be overwritten after garbage collection happens and when the memory is reallocated to a new object. This data overwrite may not happen in a timely manner, because collection must happen before the memory can be reused. Even if collection happens, the memory may never be overwritten because the program is not very active. Chapters 3 and 4 will discuss this issue and its consequences in greater detail.

Chapter 3

Measuring and reducing latent secrets in HotSpot JVM

3.1 Introduction

Managed memory runtime environments like Java eliminate many kinds of programming errors that can become security vulnerabilities. For example, Java programs are not vulnerable to buffer overflow attacks, making Java (and other "safe" languages) attractive for building security-critical software. Meanwhile, techniques such as just-in-time compilation, hot-spot optimization, and parallel garbage collection, have largely eliminated the performance penalty of using managed runtime environments. These features and a rich set of standard libraries have led to a broad adoption of Java and other such languages.

However, the HotSpot JVM introduces risk when dealing with sensitive data [14]. Our research shows that the HotSpot JVM allows session identifiers, passwords, and TLS 1.2 session keys to remain in the JVM process memory after the corresponding Java objects have been garbage collected. Furthermore, because Java provides no direct access to the underlying memory, developers cannot explicitly sanitize their sensitive data once it is no longer needed. An attacker, on the other hand, might still be able to gain access to the raw memory through many means, such as a hypervisor bypass attack, access to a swap or hibernation file, or from another process running on the same physical machine. Of course, we hope that traditional system security mechanisms can keep an attacker away from this data, but for the cases in which an attacker can gain access to the JVM's process memory, limiting the time that sensitive data remains in memory provides defense-in-depth security coverage.

Automated memory management falls into two categories: *reference counting*, which is used in Python and Swift, and *tracing garbage collection*, which is used by Java and many other language systems. Tracing garbage collection measures reachability from a set of *root objects* to every object in memory. Objects that can no longer be reached are considered *garbage*. Garbage collectors are typically *lazy*; there is a gap between when an object becomes unreachable and when the garbage collector reuses that memory. GC can also *re-arrange* the managed heap to help improve collector performance and reduce pause times. This rearrangement inherently involves *copying* objects, which can leave behind multiple "old" copies.

In this paper we demonstrate confidentiality failures due to a semantic gap between the language that programmers use, the language implementation, and the underlying execution environment, echoing similar findings in other areas (e.g. [12, 15]). Specifically, we establish the volume of secrets that an unsanitized heap can expose, using a TLS web client atop Oracle's HotSpot JVM, driven by a synthetic load under different levels of memory contention. We capture whole system memory images and then use binary string searches to find TLS keys and other sensitive data. We then make changes to the TLS code and the garbage collector in attempt to eliminate most of these secrets. We find that zeroization adds a significant workload to the JVM; Section 3.5 provides future direction on how to design JVM systems that do not sacrifice performance for improved confidentiality.

3.2 Prior Work

In 2001 Viega identified that memory is not securly deallocated in C, C++, Java, and Python runtimes [14]. Chow *et al.* showed that Unix operating systems and standard libraries failed to sanitize deallocated memory; attackers could exploit this issue to recover latent secrets from common applications like Apache and OpenSSH. The authors implemented proper sanitization in the Unix operating systems with roughly a 1% impact on performance [12, 13]. However, Chow *et al.*'s

techniques cannot address the latent secrets found in the HotSpot, because the JVM uses it's own memory management primitives. Additional work has been done to help reduce latent secrets due to shared program variables with static analysis [16]; Anikeev *et al.* [17] proposed introducing keywords into managed languages hinting at how to securely manage object instances.

Anikeev *et al.* [18] study the problem of latent secrets in an Android runtime that uses the Dalvik VM (DVM) and attempt to solve this problem by altering the GC implementation. Their work mentions negative performance impacts due to sanitization, but they do not demonstrate how well latent secrets are eliminated from the DVM heap. We investigate similar questions with two different GC implementations and measure both the performance impact and the effectiveness of eliminating latent secrets.

CleanOS [19] is the most effective solution for eliminating the *clear-text* presence of latent secrets in a VM runtime (the DVM). The researchers extended the Android SDK to allow programmers to explicitly tag some objects as *sensitive data objects* (SDOs) and developed a new GC, the *evict-idle GC* (eiGC), that properly sanitizes SDOs. Beyond programmer tagging, SDOs can be implicitly tagged as the result of taint analysis. CleanOS uses TaintDroid [20] to help identify sensitive data that are sourced from SDOs (e.g. data from TLS sockets). The eiGC protects these objects by encrypting *idle* object data with an escrowed cloud application key. When the object is idle long enough, this key is securely deallocated. If the object is still in use by the application, the eiGC can fetch the key from the cloud application and decrypt the data. This approach relies on a third party or an additional application server to manage keys in a secure manner, which is not ideal in certain situations.

The process of extracting latent secrets from dump files or system memory seems challenging, but many researchers have found the task to be quite surmountable. For example, Harrison and Xu identified RSA cryptosystem parameters in unallocated memory that had been inadvertently written to untrusted external storage as the result of a Linux kernel bug [21]. Halderman *et al.* showed

that AES encryption keys can be readily detected in RAM from their key schedule [22]. Case presented an approach for analyzing the contents of the Dalvik virtual machine [23]. Similar attacks are possible against Android smartphones, allowing for the recovery of disk encryption keys [24] and Dalvik VM memory structures [25]. Jin *et al.* used symbolic execution and intra-procedural analysis to accurately extract the composition of type data generated by C++ programs [26].

Finally, there are a variety of memory disclosure attacks and techniques. The most straightforward technique uses one process to read the memory of another process utilizing a suitable device driver or kernel module (e.g. */dev/mem* or the */proc/nnn/mem* devices). Because such devices are commonly exploited by malware, many operating systems no longer include devices for reading the memory of other processes. However, Stüttgen and Cohen developing an approach for safely loading a pre-compiled kernel modules into memory on running Linux systems [27]; their approach is now used by the Rekall Memory Forensics Framework [28].

Halderman *et al.* developed the "cold-boot attack" in which the DRAM memory from the target computer is physically chilled and then transferred to a computer that is known not to wipe memory on boot [22]. It is also possible to physically read the contents of a computer's memory using hardware that provides direct memory access (DMA). Consumer firewire interfaces, JTAG interfaces, and specially constructed interface cards can perform DMA; VöMel and Freiling survey such techniques for acquiring main memory in computers running Microsoft Windows [29]. Consequently, the threat of an attacker conducting a memory disclosure attack is significant, justifying efforts to mitigate these attacks.

We note that this class of attack may apply in a variety of different devices. Smartphones and laptops may be physically stolen or otherwise captured, giving a motivated attacker physical access to the device. Cloud services may migrate virtual machines from physical system to system, allowing for a variety of attacks while the VM is migrating, or accessing the system's memory from a potentially compromised hypervisor.

This article is predicated on the assumption that an attacker has somehow found a way to capture an *unencrypted* system memory image; based on our survey and direct experience, we believe that this threat is credible.

3.3 Measuring Latent Secrets

Here we discuss the infrastructure and software used to measure latent secrets in the HotSpot JVM. For simplicity, we used *black-box* analysis. Our Java client application repeatedly made TLS connections to our instrumented web server, creating an abundance of latent secrets in the heap. On the web server, our modified OpenSSL library recorded each session's pre-master secret (PMS) and master secret (MKB). We then searched a memory dump of our Java client's Linux virtual machine for all of the previously logged secrets from every TLS session.

We ran these experiments on a small cluster of PCs running Linux KVM; the number of simultaneous virtual machines were limited to avoid resource contention and prevent measurement discrepancies. Each experiment consisted of a pair of x64 Ubuntu 14.04 LTS VMs: our synthetic client and a TLS webserver using a modified OpenSSL library. The web server used NGINX and TLS 1.2 to serve several static web pages. The web server VMs utilized four logical cores and 2 GiB of RAM—enough to ensure that server performance wasn't the bottleneck during the experiments. The VMs that ran the Java clients were configured with 20 GiB of RAM and 4-CPUs when using the SerialGC and 8-CPUs when using G1GC. VMs running the synthetic client were rebooted at the conclusion of each experiment, allowing us to restart each run from a similar starting point.

3.3.1 Synthetic Client Functionality

Our synthetic Java client is a *multi-threaded*, *configurable* TLS web client. The client implements several parameters that manipulate the memory pressure exerted on the heap, the number of con-

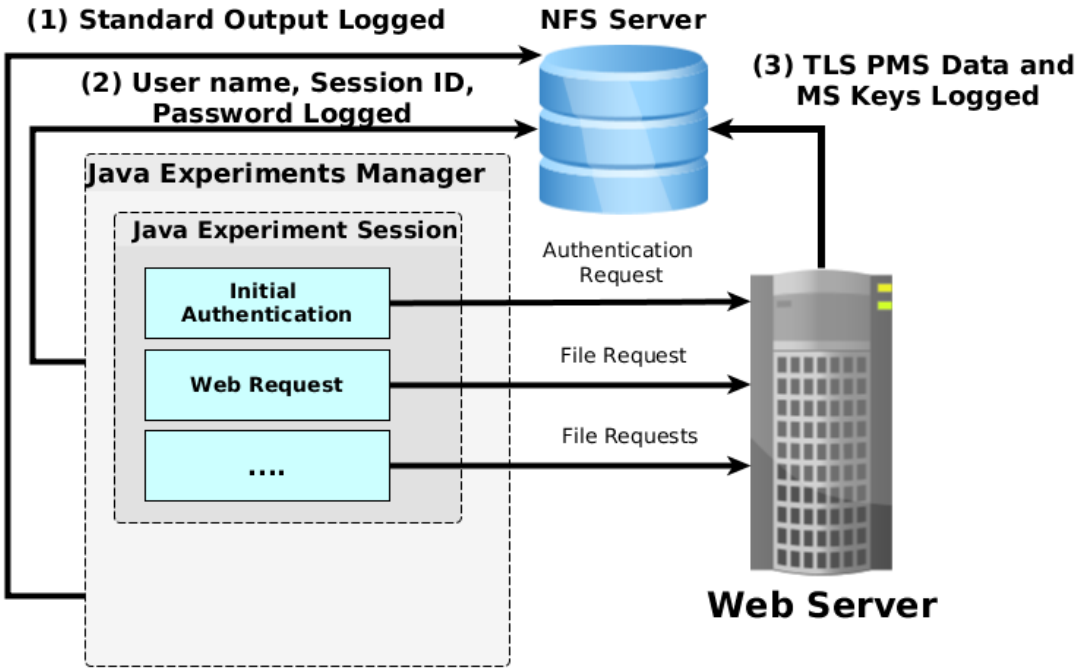


Figure 3.1 : Functional overview of our synthetic web client.

current threads, the maximum number of HTTPS requests, and the lifetime of a thread sending the web requests. These parameters provide the ability to model basic transactions for applications such as a thick- or web-service client.

For this paper we choose two specific configurations. Both configurations allowed up to 192 concurrent TLS connections that were active for at least 96 seconds. They differed in amount of heap memory allowed (e.g. memory allocated from the JVM in the form of objects) and by the number of requests allowed per thread. The high memory pressure (HMP) experiment allowed allocations to consume up to 80% of the JVM's managed memory, and the low memory pressure (LMP) experiments allowed a maximum allocation of 20% from the JVM's managed memory.

Figure 3.1 shows the two main components of the synthetic client. The Java Experiment Manager (JEM) managed all experimental sessions (threads). The Java Experimental Sessions (JES) implemented the web client functionality in a Java thread. A Python script started the experiment with

parameters that defined the behavior of the synthetic client, the IP address of the server, and where to store log files containing events and other data.

The JEM was responsible for managing the number of JESs and enforcing the experimental behavior and garbage collection parameters. The JEM controlled the number of concurrent JESs, JES allocation behavior, JES HTTPS requests, and the overall lifetime of the JES thread. Parameters controlling the garbage collection defined the frequency of collection, when to start collecting, and whether or not to pause JESs after the first GC. Our experiments also allowed us to vary the TLS library in use (Oracle vs. BouncyCastle) and whether to use the Apache HttpClient or a TLS on top of a basic Java socket.

We implemented and use three different TLS web clients in the JES. The most basic TLS client was the "Socket TLS Client". This type of client opened a TLS socket to the remote server, sent a raw HTTP request as a formatted string, received data, and closed the socket. The second client ("Apache TLS Client") used the Apache "HTTPComponents" library to create an `HttpClient`, which then connected to the remote host. Most of the internal HTTP mechanics were abstracted away, simplifying the entire retrieval task; this abstraction removed sensitive data like usernames and passwords from our control. The final client ("BoucyCastle TLS Client") was a variant of the Apache TLS Client that uses the BouncyCastle cryptography library instead of Oracle's cryptography library. This option allowed us to measure whether the TLS implementation, itself, can contribute to the volume of latent secrets.

Each implementation made every effort to remove excess references and prepare the connecting object for a future collection. In the Socket TLS Client, we close the `Socket` and set our references to it to `null` as soon as possible. The Apache `HttpClient` does not have an explicit close or shutdown API, so only references to the Apache TLS Client and BouncyCastle TLS Client be set to `null`, and we hoped its internals don't maintain references to sensitive data. We also note that the Apache `HttpClient` uses an `HttpClientConnectionManager` to manage client connec-

tions. This manager may choose to maintain open connections to the remote hosts. Such socket reuse makes reconnecting to an old peer much faster, avoiding the overhead of rebuilding a TLS connection, but may also contribute to the build-up of key material in memory longer.

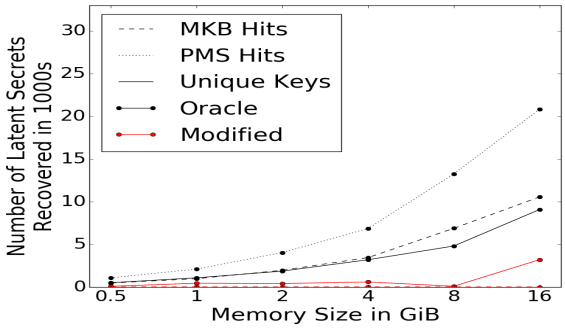
3.3.2 Memory and data analysis

Data analysis and extraction happened in three distinct phases. After an experiment, the resulting memory, TLS session data, and web client logs containing sensitive HTTP parameters such as the username and password are queued for analysis. First the analysis process scans the memory dump for latent secrets (e.g. PMS and MKBs) using `jbgrep`. This scan is conducted using two perspectives of the memory dump. The first perspective is the *raw* memory dump, which reveals all the latent secrets along with a count for each one found. The second reconstructs the process memory using virtual memory mapping, which details where the latent secret exists in the Java process (e.g., which generational heap and the address in the Java process).

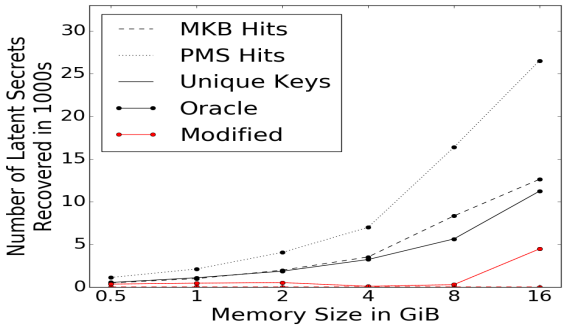
After the latent secrets are identified and counted, a post-processing step enumerates every HTTP request for each JES and pairs these requests using the TLS session data and a monotonically increasing timestamp. Although we are unable to pair the exact TLS session to the corresponding web request, such granular knowledge is not necessary to create an approximate timeline showing live objects versus latent garbage in the heap. All the dead PMS and MKB data are identified, and the results are stored.

3.4 Removing Latent Secrets

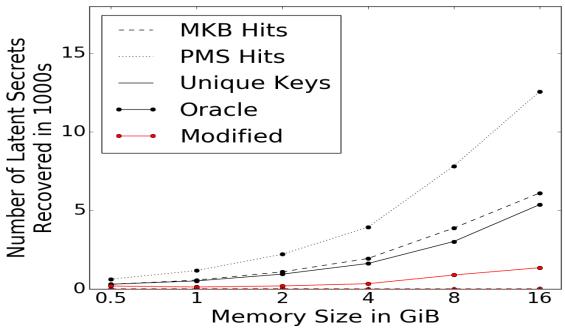
Figure 3.2 combines the results of several experiments. The preliminary experiments (shown in black) use the Oracle HotSpot JVM to examine the retention of latent secrets in the heap. These experiments use the selected GC and run with a varied heap size between 512MiB – 16GiB. The



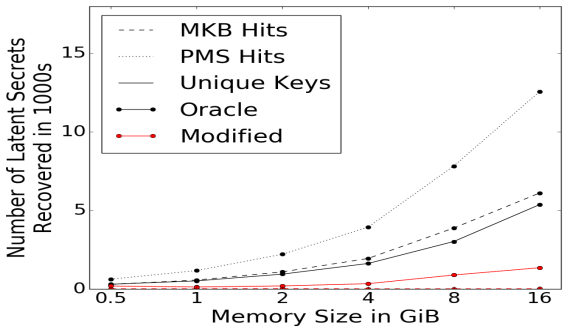
(a) Socket TLS Client with HMP parameters



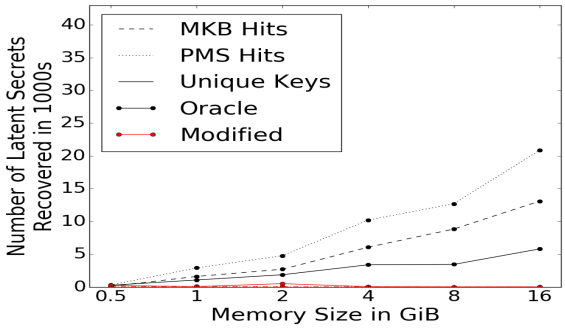
(b) Socket TLS Client with LMP parameters



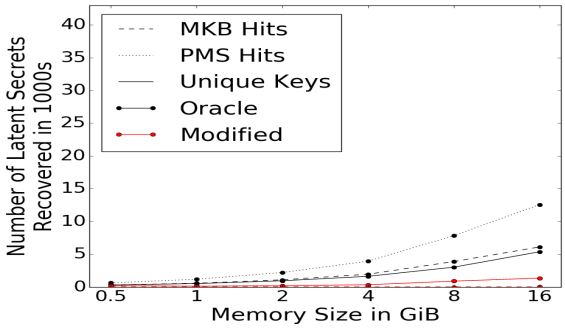
(c) Apache TLS Client with HMP parameters



(d) Apache TLS Client with LMP parameters



(e) Socket TLS Client with HMP parameters (G1GC)



(f) Apache TLS Client with HMP parameters (G1GC)

Figure 3.2 : These plots compare the results for the Socket TLS Client the Apache TLS Client. The lines show how many latent secrets can be removed from memory by sanitizing the heap space after garbage collection. High- and Low-pressure applications are also shown. Figures 3.2a-3.2a use the modified SerialGC and Figures 3.2e and 3.2f use the modified G1GC.

Heap Size (MiB)	# of TLS Sessions	Socket TLS Client Recovered Keys		Apache TLS Client Recovered Keys	
		#	%	#	%
512	5000	489	9%	286	5%
1024	5000	1059	21%	499	9%
2048	10000	1845	18%	929	9%
4096	10000	3177	31%	1608	16%
8192	15000	4786	31%	3008	20%
16384	30000	9058	30%	5354	17%

Table 3.1 : The average percentage of recoverable TLS sessions from HMP clients using the SerialGC on the Oracle HotSpot JVM.

number of TLS session also vary to establish a reasonable baseline of retained latent secrets in each heap size. We collected 20 samples per memory collection (Figures 3.2a – 3.2d) to help us identify any potential variance in our measuremnts. We see the obvious outcome where the number of recoverable TLS keys increases with heap size, doubly in some cases.

Table 3.1 shows a sample of recoverable unique keys from two control experiments. Specifically, the table focuses on the Socket and Apache TLS Clients using HMP parameters using SerialGC. The Apache TLS Client has fewer recoverable keys than Socket TLS Client, apparently attributable to the larger memory footprint of each `HttpClient`. The Socket TLS Client requires less heap memory per connection because it only requires IO buffers and a reference to the OS socket. This means the Socket TLS Client client can make more connections before GC happens.

Thus, the JVM process is a viable target for memory disclosure attacks. For each TLS key recovered in our control experiments, there are roughly 1-2 copies of the pre-master secret (PMS) data and 3-4 copies of fully intact master key blocks (MKB), i.e., TLS session keys. Multiple copies

JVM Version	Keys recovered after GC Bouncy Castle TLS Client	Apache TLS Client	Sockets TLS Client
Low Memory Pressure (LMP) Results			
Oracle JVM	1542 ± 92	2972 ± 81	1084 ± 84
Modified JVM	341 ± 55	827 ± 30	304 ± 117
Modified JVM/JCE	364 ± 102	848 ± 44	371 ± 89
High Memory Pressure (HMP) Results			
Oracle JVM	1671 ± 86	3052 ± 60	1202 ± 86
Modified JVM	406 ± 87	944 ± 78	371 ± 94
Modified JVM/JCE	375 ± 103	1010 ± 55	387 ± 56

Table 3.2 : The number of unique TLS keys that are recoverable after garbage collection.

of key data are the result of extraneous copies and excessive references to these copies. When our results refer to "unique keys," we note that an MKB can be derived from a PMS, so if we find both, we'll only count them as one "unique key."

Where are these key copies coming from? Inspection of the OpenJDK Java JDK source code reveals that local variable references are not zeroed then set to `null` and *cloned* `byte[]` values are not zeroed when they are no longer needed, so the latent data stays in memory until the memory gets reused. And, because of the generational structure of the GC, there may be additional copies of older keys.

It is clear that latent secrets are a concern. When key material from thousands of closed connections sticks around in memory, it significantly increases the risk that encryption keys might be compromised.

Consequently, we devised two approaches to address this issue, both requiring changes to the OpenJDK source code. First, we attempted to patch the Java Cryptography Engine (JCE) and Java Secure Sockets Extensions (JSSE). After manually auditing the code, we took steps to ensure classes perform explicit sanitization on local variables containing secrets, and explicit calls are added to ensure key data is overwritten when TLS sessions and sockets close. Our second approach focused on modifying the JVM internals. Specifically, we added code to zero memory as it was de-allocated, and to zero all unused heap spaces after each GC-cycle.

3.4.1 Adding Sanitization to the JVM

We modified the OpenJDK HotSpot JVM source code to implement a *global* sanitization solution in the heap. For simplicity, we choose to modify the SerialGC and G1GC implementations—the current and future default *server* garbage collectors for the HotSpot JVM. First, we focused our efforts on cleansing the young generation in the SerialGC, and then we tackled the problem in the tenured generation. The following approach generalizes nicely to both collectors. The emphasis of the approach forces sanitization on the internal memory structures of the JVM and managed heap during and after the garbage collection cycle happens.

Since generational GC partitions the heap, the algorithms and policies used to collect each generation can vary. For example, the SerialGC young generation uses copy-collection while the tenured generation generally relies on mark-and-sweep followed by compaction. The G1GC employs similar techniques to decide how and when incremental or full collection happen. In both cases, we tagged along with the sweep or incremental phases, zeroing out regions of the memory corresponding to dead objects. When compaction happens, we similarly zero out the original objects, one by one, after they're relocated. (Unfortunately, Hotspot's SerialGC doesn't ever do a giant copy-compaction during collection in the tenured space with a from-space and a to-space, so there's never a huge block of memory we can blindly zero out.)

Zeroing individual objects, or arrays, as the sweep phase or copy phase figures out that they're garbage, seemed like a relatively efficient change to make to the garage collection, since the memory in question was just recently touched, so it should already be in the CPU's cache. Unfortunately, this strategy required us to understand all of the specific tricks that the garbage collector uses, so we know when it's truly safe to write zeros into memory.

Notably, we encountered cases where *invalid* dummy objects were placed in the heap. Without knowing this fact, we would check pointers and class types using internal APIs, and these checks caused segmentation faults in the JVM. After some investigation, we discovered that this issue was the result of a hack to make the heap appear to be contiguous during collection, which is a precondition to make GC work correctly. Recall, each TLAB is a small partition of the eden space, so the JVM fills the empty spaces with dummy objects during GC or when a TLAB is invalidated. We resolved this issue by ensuring `Klass` pointers (i.e., pointers to the C++ representation of a Java class) fall inside the Java metaspace, where all Java meta-objects (e.g. classes, methods, etc.) reside, prior to overwriting.

We also had a variety of other minor issues. For example, dealing with primitive Java types (like `byte` arrays) versus class types (like `Byte` arrays) required specific logic.

After modifying the garbage collector, we still found latent secrets that survived, and they were outside of the managed heap. Recall that the JVM maintains memory blocks that are explicitly allocated and freed. Latent secrets were getting copied there as well. We addressed the problem by sanitizing all internal memory deallocations (similar to [12]). Leveraging the JVM's native memory tracking (NMT) for this task. Typically, NMT is used to track internal memory allocations to help with profiling, diagnostics, and debugging. For our purposes, we used NMT to identify the size of each allocation, and then we zero the buffer before the memory is returned to allocation pool.

3.4.2 Sanitization Effectiveness

Our second set of experiments focus on assessing our JCE/JSSE modifications, and we also explore whether Oracle's cryptography library might be responsible for retaining additional latent secrets. Three different Java runtime configurations use the SerialGC: Oracle HotSpot JVM, modified OpenJDK HotSpot JVM and modified OpenJDK HotSpot JVM cryptography libraries; our three different TLS clients run in these environments. The JVM heap size is fixed at 4GiB, and 10 memory dumps are collected for each experimental configuration. To ensure that sanitization is working properly, the JEM pauses the JES threads and performs an explicit GC to invoke the added sanitization steps before dumping the VM's system memory. The Socket TLS Client experiment averages 11.9K TLS sessions, and the Apache TLS Client and Apache TLS Client with BouncyCastle experiments average 16.6K and 16.8K respectively.

Table 3.2 shows the average results from this set of experiments. The modified JVMs exhibit a significant drop in the number of latent secrets present. However, this massive reduction is mostly attributable to the sanitization added after collecting the young generation. Since sanitization of the heap generations depends largely on allocation failures, the tenured generation needs more collection activity to trigger the removal of latent secrets, which we realized after analyzing the GC logs.

We also see that the JCE and JSSE modifications modestly increased the number of latent secrets in the heap. We're not entirely sure why this occurred. It's possible that our code modifications created side-effects on the JIT compiler. For example, compiler could have optimized out our zeroing code or perhaps the resulting code maintains unnecessary references to otherwise dead objects. These negative findings reinforce the importance of support from not only the garbage collector but also the underlying JVM. Pure application-level zeroing of data will never adequately address the problem.

Both the "BouncyCastle" configuration and the "Apache" configuration use the same Apache HTTP client library, so the only significant difference is that the "Apache" configuration is using the Oracle TLS library. Why is the BouncyCastle version so much better? A manual inspection of the BouncyCastle code shows that the authors make fewer copies of key material. That said, the "Socket TLS Client" experiment drops the Apache HTTP client library and directly drives the Oracle TLS libraries. This gives up the performance and concurrency features of the Apache library, but has the fewest latent secrets. These results suggest that complex interactions between libraries and networking layers can have unforeseen increases in the volume of latent secrets.

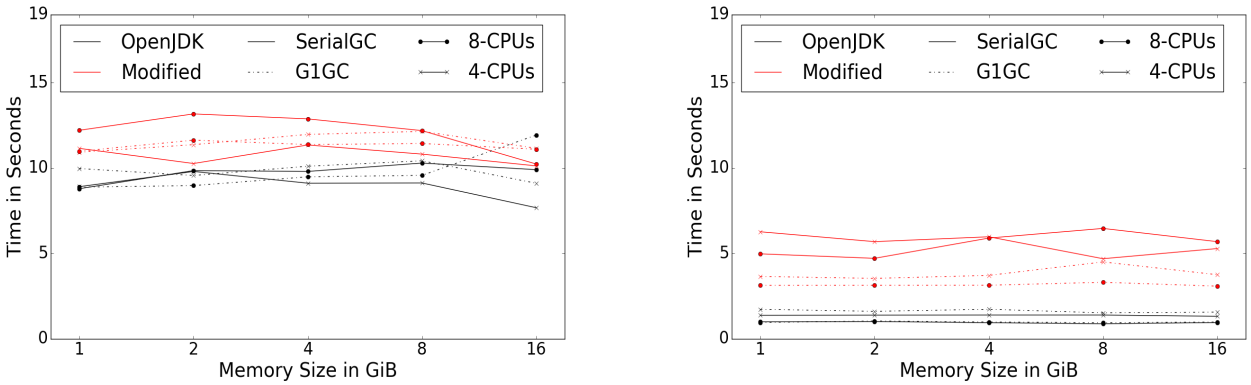
The third and fourth experiments recreate the conditions from the initial assessments to evalutate the overall reduction of latent secrets. The JVM heap size varies from 512 MiB – 16 GiB, the number of sessions vary between 5K – 30K, and the host system uses 4-CPUs (SerialGC) or 8-CPUs (G1GC). In the third experiment, the focus is on both the HMP and LMP configurations of the Apache and Socket TLS Client, and the fourth experiment only looks at HMP configurations.

Figures 3.2a-3.2d presents a progression towards eliminating latent secrets in the heap using the SerialGC. In some circumstances the volume of latent secrets stays small regardless of heap size, while in other circumstances the volume of latent secrets starts small, but with very large heaps it grows significantly. We believe this is a consequence of the tenuring process. The GC may not collect seemingly dead objects in the tenured generation because extraneous references to those objects are not dead yet. Additionally, if memory pressure is inadequate then this generation may never be collected.

Figures 3.2a and 3.2c demonstrate that this tenuring effect on objects is mitigated, and most, if not all, of the latent secrets are eliminated. The G1GC offers improved performance and sanitization because it uses smaller heap regions to store objects. These partitions help facilitate parallel GC, and the smaller regions can lessen the amount of memory that needs to be zeroed during each GC, especially in the tenured generation. After some initial experiments, we found that *two* explicit

GC calls are necessary to achieve *full* heap sanitization; the SerialGC required *four* explicit calls to fully zero the heap. The first explicit GC triggers an incremental collection, and the second forces a full collection because it interrupts the incremental collection.

3.4.3 Benchmarking



(a) `tradebeans` - DayTrader Benchmarks

(b) `lusearch` - Text Searching Benchmark

Figure 3.3 : The benchmarks show that our modifications have significant effects on the GC performance. The unmodified OpenJDK (black) benchmarks are the baseline.

Overall, the findings from each of the experiments demonstrate the difficulty of eliminating sensitive data in a managed runtime. Now, we want to determine how our modifications affect the JVM's performance. We use the Dacapo benchmark suite, version 9.12 [30] to measure these impacts. This benchmark framework uses several real world applications to measure runtime performance, but we only show two for brevity: `lusearch` and `tradebeans`. The `lusearch` benchmark performs a number of searches over a textual corpus using `lucene`, a text searching engine. The `tradebeans` benchmark is a DayTrader application that interacts with an Apache Geronimo backend and `h2`.

Each benchmark executes 50 times with the default workload parameter, *pre-iteration* GC disabled

and four workload threads. The benchmarks run on an unmodified OpenJDK HotSpot JVM and the modified JVM with variable heap sizes (e.g. 1 GiB–16 GiB) and CPUS (either 4 or 8). The unmodified JVM is built on the same machine using the same build settings of our modified JVM; we take this precaution to eliminate any build or code optimization variables that might influence the benchmark results, which cannot otherwise be done with Oracle's HotSpot JVM.

Figure 3.3 shows the benchmark results, which are not surprisrising. The G1GC with 8-CPUs incurred performance penalties from sanitization: 200% in `lusearch` and 21% in `tradebeans`. We discuss how future GC implementations can address the performance issue in §3.5.

3.5 Discussion and Future Work

Cleansing latent secrets from managed memory is a challenging problem, and application or runtime demands are going to dictate how these challenges are addressed. We have seen that parallel collectors like the G1GC offer some relief for these issues, but redesigning the JVM to provide for proper sanitization seems to be a better alternative in terms of performance. CleanOS exemplifies how these modifications to both the managed heap and the software VM imprpove security [19]. This extension to Android encrypts sensitive data objects in place, and then when the object is no longer needed or it's idle, the key is securely deallocated. However, when considering server-side changes, re-engineering the JVM should also consider dealing with sensitive native IO operations, shared variables in applications [16], and incorporating explicit *data lifetimes* into the programming language [13, 17].

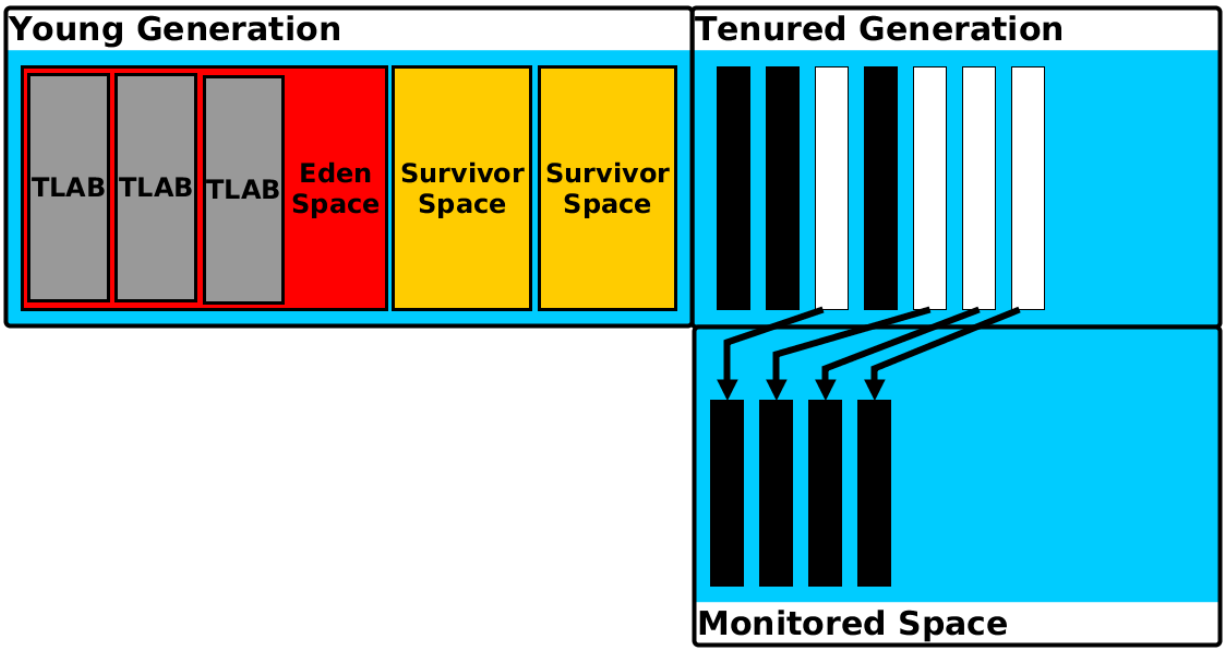


Figure 3.4 : A generational heap layout concept that uses a monitored space where *sensitive* data is stored and maintained.

A feasible strategy explicitly segregates sensitive data into its own *monitored space* that identifies and removes latent secrets promptly. Developers need the ability to define data lifetimes when writing their code. Java currently has meta-data tags, known as *annotations*, that help with the compile-, build-, and runtime operations. *Data lifetime* annotations could help the JVM handle, store, and sanitize these data items without impacting other code.

Figure 3.4 shows a hypothetical heap with an additional monitored space. In the monitored space, memory might be explicitly reference counted, allowing for immediate sanitization when an object dies. Furthermore, these objects could have explicit "destroy" APIs, so applications can explicitly kill them or an executive task can reclaim the object. Functionally, references from the main heap to the monitored space would act like *weak references*, making it clear to the application author that sensitive, monitored data could disappear at any time, and must be checked explicitly before each use. Such a strategy sounds straightforward, but it would have a variety of problems. For example, sensitive data can touch multiple layers of the protocol stack. Zero-copy IO techniques

(e.g., IO-Lite [31]) could help with this, but require the entire stack to be engineered around a particular buffer management strategy. Changes like this would break existing APIs and require re-engineering of libraries.

3.6 Conclusion

Java and the HotSpot JVM will likely be around for decades to come. This runtime offers a rich set of development tools and libraries that help engineers construct and deploy useful software. However, servers and services are susceptible to a number of attacks through a variety of vectors, so there is no guarantee that the system where the software executes will remain free from compromise. Attackers evolve quickly, and they will realize that the JVM does not effectively sanitize internal or Java heap memory. This lack of sanitization can compromise sensitive data and lead to unforeseen impacts and consequences. We have taken a proactive approach to this problem by measuring its existence and developing several strategies to help mitigate the problem.

Problems with managed runtime environments like Java and the JVM are well known, but they are not well understood. Our research provides several fundamental elements. We establish the heaps capacity to retain latent secrets. Furthermore, we show that as heaps increase in size the number of latent secrets also increases. Cryptographic libraries should protect sensitive data such as keys, but we find that Oracle's JCE implementation of TLS 1.2 does not attempt to eliminate key data.

Given the lack of sanitization in the Java heap, we demonstrate several approaches that reduce the accumulation of sensitive data. Used together, the number of TLS keys are reduced dramatically. To accomplish this feat, we first modify the JVM to zero unused heap space in the young generation. Second, the tenured generation is also wiped when the dead objects or live objects are encountered during the mark-sweep-compact collection algorithm. We also zero unused heap space after the garbage collection, showed these approaches work for both the SerialGC and the

G1GC.

We define how to improve performance of garbage collection implementations while keeping data security in mind. Our proposed design modifies the overall structure of the heap, carving out a segment specifically for sensitive data. The design also exploits Java annotations, which can be used to inform the runtime about how to properly handle specific types of data. This design keeps execution and runtime efficiency in mind while allowing for the timely and effective sanitization of data.

Chapter 4

ReCOOP: Framework for recovering live and dead Java objects in HotSpot JVM

4.1 Introduction

Memory analysis can yield important information when performing forensic analysis as a part of incident response, but it can also be extremely tedious. Several factors hinder memory forensics. First, an analyst requires tools or some understanding about how to extract and interpret the data structures supporting the program. Second, these data structures might be incomplete, overwritten or missing. Finally, the amount of data extracted from memory and its creation order can be impossible know for certain.

Standard memory analysis frameworks like Rekall and Volatility focus on recovering forensic information from OS structures and services. Conversely, when dealing with a garbage-collected / managed runtime memory system, the interpretation of recovered memory objects depends not on the host machine's architecture or operating system, but on the particularities of the managed runtime implementation. As more applications are written in language like Java, Python, or Microsoft's .Net languages, using garbage collection to manage their memory, threads, and other system state, it becomes increasingly important for forensics tools to address these systems.

Furthermore, attackers are increasingly crafting exploits for code running within managed runtimes, delivering code-injection attacks against a variety of services. Forensic tools must then connect low-level kernel state with high-level object state to present a coherent picture of the attacker at work.

This research builds on our previous work exploring JVM data retention through the observation and measurement of latent artifacts in the heap [32]. Here we present JVM tools that we built to rapidly analyze an obfuscated malware. Next, we demonstrate that evidence of sockets and other important artifacts can be recovered from residual data in the Java heap, even if they are not present in the operating system. We focus on the HotSpot Java Virtual Machines because it has been widely adopted within the enterprise and is a vector of current attacks against a number of industries.

The remainder of this paper is organized as follows: Section 4.2 focuses on related work and past research, and Section 4.3 talks about the process of recovering Java objects and low-level object pointers (e.g. Original Object Pointers (OOPs)) from the HotSpot JVM. Section 4.4 shows how this can be applied to memory forensics and malware analysis. Section 4.5 expands on future needs for this project. Section 4.6 concludes.

4.2 Prior work

A large body of work has established usable techniques for copying memory, including [22]'s "cold-boot attack," direct memory access (DMA), FireWire, JTAG, and specially constructed interface cards can perform DMA; [29] survey such techniques for acquiring main memory in computers running Microsoft Windows.

The two most common forensic frameworks for decoding memory dumps are the Volatility Memory Forensics Framework [33] and Rekall Memory Forensics Framework [28]. These frameworks are written in Python and implement plugins for specific functions such as listing valid processes and open network connections. Separately, researchers have demonstrated special-purpose memory analysis tools for rendering the pieces of documents that remain in an application's memory [34], recovering Android GUIs from apps [35], and recovering photographic images that were

shown in the view finder, even if they were never written to storage [36].

[14] identified that memory is not securely deallocated in not only C, C++, but also in systems with managed runtimes, such as Java, and Python, potentially allowing sensitive information to be recovered. [12, 13] showed that Unix operating systems and standard libraries failed to sanitize deallocated memory; attackers could exploit this issue to recover latent secrets from common applications like Apache and OpenSSH.

[37] shows that sensitive information from the Python runtime is easily retrievable, and Java has similar issues. Forensic analysts can potentially recover copies of Java objects long after the active objects have been garbage collected and overwritten. Such objects might contain sensitive data related to service and user accounts, financial data, or network artifacts left behind by attackers. Forensic analysts can use such objects for establishing an event timeline, looking for evidence related to compromises, understanding the behavior of malicious software, and enumerating compromised data. [32] showed that the JVM fails to overwrite garbage-collected objects, potentially allowing the recovery of TLS secrets long after the TLS connection has been terminated.

Many researchers have demonstrated techniques for recovering usable latent secrets from dump files or system memory. For example, [21] identified RSA cryptosystem parameters in unallocated memory that had been inadvertently written to untrusted external storage as the result of a Linux kernel bug. [22] showed that AES encryption keys can be readily detected in RAM from their key schedule. [23] presented an approach for analyzing the contents of the Dalvik virtual machine. Similar attacks are possible against Android smartphones, allowing for the recovery of disk encryption keys [24] and Dalvik VM memory structures [25].

Automated recovery of memory structures is the pinacle of forensics work. This means that source code is not longer necessary to create the data structure required to interpret program memory blobs. Lin *et al.*showed that dynamic execution traces from programs could be used to recontruct protocols and data structures [38, 39]. Jin *et al.*also used symbolic execution and intra-procedural

analysis to accurately extract the composition of type data generated by C++ programs [26]. Dolan *et al.*and Lin *et al.*show that even though a program and kernel space may be unpredictable, data structures signatures can be very useful for finding and extracting relevant data [40, 41].

Data carving is also an important part of memory and file system analysis. Carving is an activity whereby files and content are reconstructed using signatures and other data. Richard and Roussev use file headers and footers find and extract file content in a parallel manner [42]. Beverly *et al.*apply data carving to memory as a means of finding network content and relevant data structures. Hand *et al.*use data carving to find and extract binaries from memory, which is especially challenging. Unlike the files found on disk, the runtime executable representation transforms and relocates various file sections throughout memory.

This article is predicated on the assumption that an attacker or forensic examiner has somehow found a way to capture an *unencrypted* system memory image; based on our survey and direct experience, we believe that this threat is credible.

4.3 Approach

Our memory analysis approach focuses on both the virtual machine and the managed memory. Our analysis components rely on a simple overlay system for data structure interpretation and a simple system for accessing memory using the process's virtual addressing scheme. Our analysis framework, RecOOP, is written in Python and can be used with an interactive environment like IPython [43] or as a library like Rekall.

Figure 4.1 depicts the process we use to recover objects from managed memory. Currently, our RecOOP analysis focuses on recovering HotSpot JVM OOPs from x86 architectures. Adding support for 64-bit machines would only require minor modifications to address the OOP encoding. We similarly expect that our work would generalize to support other managed runtime environments

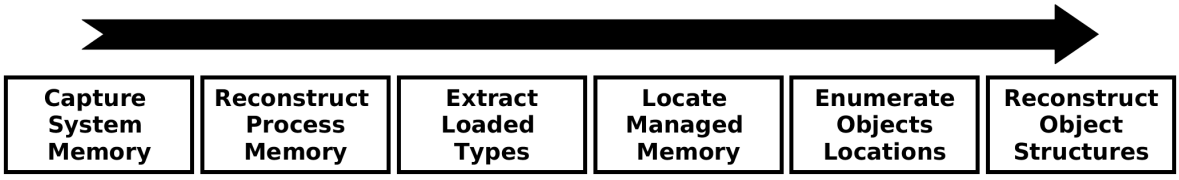


Figure 4.1 : An overview of the steps that RecOOP takes to extract and recover managed memory objects for forensic analysis.

such as those used by Mono, .Net, or JavaScript.

We implemented our analysis for the Linux and the Windows operating systems. We have successfully tested RecOOP against 32-bit versions of Ubuntu, Windows XP SP3, Windows 7, and Windows 8 with a Java heap size of 2GiB. Overlays are structural templates used to interpret raw memory as a program data structure. Only 8 out of 150 C++ overlays require different padding to achieve the correct memory layout on the different OSs. We believe these differences are due to compilers, which tend to vary field padding in the structures.

4.3.1 Process Reconstruction

RecOOP analysis begins with process reconstruction. If the process's memory has not already been extracted from a RAM image, RecOOP will dump it using the Volatility Framework. Volatility will identify the target process by name or PID and enumerate all the physical memory page frames, which are then ordered according to the process's virtual address space. Finally, the memory is saved to file for future analysis or for use with other tools such as Radare [44].

4.3.2 Extract Loaded Classes

Program portability in managed runtimes is accomplished through several key systems that *resolve*, *link*, and sometimes *compile* the program being loaded and executed. Internally, there is a loader and type system used to find and load specific classes or types, and then store these types for future reference. The loader will look inside the application or loading path to find the correct library.

When the required types, classes, and code are loaded into the virtual memory, symbols for each of these artifacts are created. Once completed, the runtime then *links* together the code for each of the classes for the given types. Linking ensures that all class dependencies for inter-class method calls and field access are loaded. Linking also optimizes method calls in classes that implement an `interface`. For example, if `class Foo implements Boingo`, the links to the `Boingo` methods need to be created to reduce any performance penalties when `Foo` is treated as a `Boingo`. After linking and loading, the original class file defining the Java types and code are transformed into machine optimized structures. These transformations are with their symbolic references in a central location.

The HotSpot JVM stores requisite information in three different hash tables: a `SystemDictionary`, `a SymbolTable`, and a `StringTable`. The `SystemDictionary` contains all the loaded type information (e.g. Java classes). The `SymbolTable` contains all the loaded symbols for classes, methods, fields, and enumerable types. Finally, the `StringTable` contains all the constant strings or strings that exist for long periods of time. Generally, only the types required for linking are resolved and loaded into the runtime; this proves useful when with dealing obfuscated JAR files, because forensic or malware analysis need only focus on the loaded class files and types.

Our JVM analysis engine first looks for the symbol table and then the system dictionary. The symbol table is a good place to begin, both because it's structurally simple and because those strings will be helpful to us later.

Memory		Data Structure Interpretation of the
Address	Values	`SystemDictionary::_dictionary`
`0x00e5b928`	**0x00004e2b**	[struct_field: int table_size]
`0x00e5b92c`	`0x00e5cd50`	[struct_field: SymbolTableBucket* buckets]
`0x00e5b930`	`0x00000000`	[struct_field: SymbolTableEntry* free_list]
`0x00e5b934`	`0x14283408`	[struct_field: char* first_free_entry]
`0x00e5b938`	`0x14283be0`	[struct_field: char* end_block]
`0x00e5b93c`	**0x0000000c**	[struct_field: int entry_size]
`0x00e5b940`	**0x00002f4d**	[struct_field: int number_of_entries]

Table 4.1 : The memory layout of a JVM `SystemDictionary` captured from an embedded Windows 7 OS instance.

These data structures are located by scanning for invariant values (**0x00004e2b** or **0x000003f1**) in the C++ `_table_size` field of the structure. When these values are found, `_entry_size` and `_number_of_entries` are used for an initial sanity check. The `_entry_size` is the size in bytes for each value entry (e.g. 1 entry = 0x000C bytes). We place an upper bound on `_number_of_entries` that starts at 100K entries but can be adjusted if necessary. Table 4.1 shows the memory layout of a system dictionary that we want to apply these constraints to.

When these constraints are met, the engine attempts to parse a subset of the hash table entries. The system dictionary has a pointer to these entries (e.g. `HashTableBuckets* _buckets`): the internal array that forms the spine of the hashtable. The engine iterates over this array and tries to follow the bucket entries to the target value using a memory overlay. If the entries can be interpreted as the expected values, we accept it as valid. For example, the C++ type `Symbol` is a `vtable`, followed by some metadata, size, and the symbol string. As a heuristic, if the length of this string exceeds a manually-chosen threshhold, the entry is considered invalid.

Memory			Data Structure Interpretation	
Offset	Address	Values	Value Information	`SystemDictionary::_dictionary` entry values
0x00	0x142d61b4	0x0b32967d		[struct_field: int _hash]
	0x142d61b8	0x00000000		[struct_field: DictionaryEntry* _next]
	0x142d61bc	**0x13fdc908**	Klass*: java/nio/channels/ByteChannel	[struct_field: Klass* _literal]
	0x142d61c0	0x00000000		
	0x142d61c4	0x00e5cd18	[→ oop class_loader]	
0x14	0x142d61c8	0x257f6796		[struct_field: int _hash]
	0x142d61cc	0x00000000		[struct_field: DictionaryEntry* _next]
	0x142d61d0	**0x13fdccb8**	Klass*: java/nio/channels/SeekableByteChannel	[struct_field: Klass* _literal]
	0x142d61d4	0x00000000		
	0x142d61d8	0x00e5cd18	[→ oop class_loader]	
	0x142d61dc	0x55f713ed		[struct_field: int _hash]
0x28	0x142d61e0	0x00f357f8		[struct_field: DictionaryEntry* _next]
	0x142d61e4	**0x13fdcf58**	Klass*: java/nio/channels/GatheringByteChannel	[struct_field: Klass* _literal]
	0x142d61e8	0x00000000		
	0x142d61ec	0x00e5cd18	[→ oop class_loader]	

Table 4.2 : A memory dump showing the offsets and values embedded Windows 7 OS instance.

Table 4.2 shows some entries from a valid dictionary found by the JVM analysis engine. Most `Klass` structures should have symbol names appearing in the symbol table, so when we parse candidate dictionaries, the dictionary entries (e.g. `Klass *`) names are checked to see if they are known symbols. If a majority of these symbols are found, we accept the candidate. The product of this analysis yields the low-level memory layout of each Java class, methods, and other meta-data like the Java constant pool. (Note: since the JVM supports class unloading, unloaded `Klass` names may not be present in the `SymbolTable`, even when dead objects of those types are still in heap pages waiting to be reused.) This extraction technique is OS agnostic and easily automated.

Alternative approach: the JVM tool interface Prior to developing these techniques, we attempted to use symbol structures intended for the JVM tool interface (JVMTI). These structures were found using string pointers to the symbol names, the structure size, static values in the fields,

and the location relative to the JVM library's base offset. When the JVM is started, these structures are filled with the appropriate runtime data structures (e.g. `SystemDictionary`, `SymbolTable`). We also used an optimization technique to find the *best fit* memory locations based on the locations of other recovered structures, the order of the structures, and invariant values that should be present in the structure.

Unfortunately, this approach has many obstacles. First, it was overly complex; the identification of strings and reverse-mapping them to pointers was time intensive, and the specific strings and structures were not always present in memory. Second, every version of Java requires a new set of *constraints* because the location of the JVMTI symbols and data structures change. Thus, this approach could not generalize across multiple platforms and JVM versions. Finally, since these structures and JVMTI symbols were not in use, the OS paged the sections of the process's virtual memory out to make space for other relevant data. Most memory analysis protocols do not consider the OS swap or page files; thus, if RAM was dumped near the start time of the JVM process, recovery of the dictionary, symbol table, and string table addresses were likely, but after a few minutes the chance of success dissipated.

4.3.3 Identifying Managed Memory

Knowing the location of managed memory helps with object enumeration and with sanity checking whether or not the results are valid. However, automatically and correctly identifying these segments is difficult. We err on the side of caution and enumerate all possible locations. To prevent erroneous object identification, we perform type checking on every object's non-primitive field references.

Potential managed memory areas are found by looking for an abundance of type-pointers (e.g. `Klass*`). Every object is required to have a defined type. Consequently, areas with a large number of type-pointers are likely to contain objects. The exceptions to this rule are places where class metadata

GC Log Message		
Generational Space		Start and End of the Space
eden space	*[...]* used	[`0xa4800000`, *[...]* `0xa4c50000`)
from space	*[...]* used	[`0xa4c50000`, *[...]* `0xa4cd0000`)
to space	*[...]* used	[`0xa4cd0000`, *[...]* `0xa4d50000`)
the space	*[...]* used	[`0xa9d50000`, *[...]* `0xaa800000`)

Table 4.3 : The regular expression "`space.*used`" used in conjunction with `ffastrings` to determine the eden, survivor, and tenure generation spaces. Note *[...]* signifies omitted message content.

or compiler interface data structures are located. Most of the class metadata is known, so these addresses are filtered out. For other areas of memory, we rely on our type checking to remove invalid entries.

We isolate managed memory boundaries by first ignoring all memory regions less than 256KiB, since this is less than the smallest default heap space. Second, we only consider pages with more than 10 type-pointers, and then we smooth variations using a moving average. We only consider areas with more then 10 type-pointers in at least 32 consecutive pages (e.g. $32 \times 4096\text{B} = 64\text{KiB}$). This algorithm might need adjustment for G1GC, because G1GC uses *humongous* memory regions (e.g. large allocations exceeding multiple MiB) for large objects.

This analysis only establishes boundaries where objects might be clustered. We avoid identifying memory regions as a particular generational space (e.g. eden, tenured, etc.), as this could lead to misclassifications. For example, the JVM may expand or contract a heap space depending on application activity. Using our analysis, we might misclassify two segments of memory as different spaces, when they were part of the same region at some point in time. A consequence of

Adwind Obfuscated Java Malware on Linux

Loaded Classes = 1626

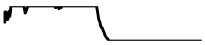
Address Range	Type Pointers	Unique Pointers	Pointer Occurrences Per Page (Y-axis: 0-64)
0xa32de000-0xa3355000	1353	265	
0xa33ce000-0xa349d000	2735	331	
0xa349e000-0xa34f5000	609	122	
0xa3600000-0xa3692000	362	360	
0xa40b0000-0xa4779000	11926	1229	
0xa47ff000-0xa4c0f000	13261	266	
0xa4c50000-0xa4c92000	129	28	
0xa4cd0000-0xa4d50000	1121	79	
0xa9d50000-0xaa000000	28810	661	
0xb6936000-0xb6996000	427	413	
0xc0001000-0xf7bfe000	11085	1211	

Table 4.4 : Number of pointers found in address ranges with more than 10 unique “type-pointer” occurences found on addressable word boundaries in a version of the Adwind malware. The *red*, *yellow*, and *black* lines correspond to the eden, survivor, and tenure space, respectively. Table 4.3 shows how these locations are found.

this misclassification might also lead us to miss regions where Java objects are present.

The JVM is very good about measuring performance and logging events, so if identifying generations is necessary, it still might be possible to do so without using type-pointers. If at least one GC event has occurred, the JVM logs information about all the heap spaces internally. These log strings contain the named heap space, start and end addresses, and other information. These messages can be found by searching a strings dump using regular expressions like "`space.*used|Metaspace.*used`." This expression will reveal most, if not all, of the managed memory spaces used by the Java heap.

To demonstrate these procedures, we analyze data from the Adwind malware analysis case study in the next section. Table 4.3 shows the most relevant information with an emphasis on the heap space and memory region. The color of the heap space is also reflected in Table 4.4, which shows the results of the type-pointer clustering. Using type-pointers narrows the number of memory regions that need to be scanned from 431 to 11, and it isolates the areas where objects might exist. The sparklines show several memory chunks that contain some of these regions, most obviously in the `0xa47ff000-0xa4c0f000`, `0xa4cd0000-0xa4d50000`, and `0xa9d50000-0xaa000000` memory chunks. If we want more granular heap information, additional memory analysis is required.

4.3.4 Enumerate and Extract Objects

The location of type-pointers is used to help object enumeration and extraction. Enumeration for objects like threads, sockets, and files happens automatically, but RecOOP permits enumerating specific types of objects any time after the managed memory is identified. In the previous phase, all the addresses to type-pointers were found and saved. These addresses are used to extract objects if they fall in a managed memory boundary.

Object extraction happens in several phases. First, we check that the address adheres to the basic object structure. Next, we use the loaded type information to determine the size of the object and

locate its references. Then, each non-primitive field is parsed recursively, repeating these steps. The field references are checked to see whether the value is `null` or the given type of the field. Note, we also track all the potential classes a reference could be due to polymorphism. After the fields have all been parsed and extracted, all the values in the fields are updated, and the process completes. Values are set after all the referenced objects are enumerated to avoid an uncontrolled recursion.

Java threads (e.g. `java.lang.Thread`) are enumerated and extracted first. During this process, the native structures implementing the thread are also identified and mined for information. After the initial identification, each thread is checked for validity and fields holding pertinent information are analyzed, most importantly `eetop`, the thread's native address. We use this field to find the linked list containing all the threads, and we iterate over it to identify any missing threads from the Java heap. If any are found, we repeat the object analysis for the missing thread.

Buffers and streams are investigated next because they are typically used to manage IO between the program, the JVM, and the operating system. Given the ubiquitous nature of these objects, there are a number of base and abstract classes (e.g. `java.io.InputStream`) that are used to create the different IO classes like `java.io.BufferedReader`. We were challenged by the polymorphism and the number of types an object might implement. For example, determining if a `SocketInputStream` is used by a `java.io.BufferedInputStream` requires identifying the `java.io.BufferedInputStream` that wraps the `SocketInputStream`. To deal with this issue, we perform multiple scans for the different IO implementations and create a basic link table. This link table helps cut through obscure object relationships to map IO classes with buffers and other data. Generally, we found that the only IO classes containing buffered data were either used by a buffered IO class or the class maintained its own buffer, as was the case with `InflaterInputStream`.

Native buffers are used to marshal IO data in and out of the JVM. Classes used for the functionality

appear to implement the `DirectByteBuffer` interface, which permits direct memory access. We have only found the implementations `MappedByteBuffer`, `NativeBuffer`, and `HeapByteBuffer` in the source code. Data in these buffers is captured, but it is volatile and may not be useful.

File information is collected from objects using the `java.io.FileDescriptor` or `java.io.File` type. For the most part, the filename or path are the only useful information found in this object type. If there is a reference from an IO object like a `FileChannelImpl` or `FileInputStream`, we might be able to determine whether or not the file is open. If buffering is not used by the IO stream, identifying any attributable data is difficult.

JAR files and entries contain information related to loaded files and might reveal sensitive information by way of compressed streams. JAR files typically hold all the program resources and class files for a library or program. Class files are decompressed and loaded from JAR files as a `ZipEntry` object. Usually, decompression and loading happen in lockstep, so any data related to the process may dissipate very quickly. When raw compressed data is present in memory, a *zlib* library may be able to decompress it. We have been able to successfully recover JAR filenames, named entries, and decompressed entries. If parts of the JAR file are present in memory, we read the low-level zip file structure, dump the resident data, and investigate the result as a zip file.

Socket objects can reveal connections well after the artifacts disappear in the OS. In particular, the IP address along with the remote and local port are extracted, if the object is still intact. We also attempt to associate the socket with any identified streams and data buffers.

Child process information is collected from the `ProcessBuilder` and OS `Process` implementation class. The `ProcessBuilder` class is the typical way to start a process. This class takes a command string or an array of strings in addition to any object for redirecting IO. Once the process is started, an OS-specific implementation of a `Process` object is created. Unfortunately, when GC happens, the string objects used for the command string are likely to be overwritten. These overwrites can prevent identifying the command by name.

The `Process` object remains in the heap for a significant period of time. In our experiments, even though GC happened several times, all `Process` objects were still recoverable. Additionally, the IO buffers `stdout`, `stdin`, and `stderr` retained some data. Even though the information used in the original instantiation of the process dissipated, we could use the process output to identify some of the processes.

One of the benefits of a managed runtime for forensic analysis is event ordering. In the HotSpot JVM, memory is allocated from the heap or TLABs directly after the last allocation; this sequential allocation is a fundamental property of a wide variety of garbage collection strategies. Because the TLABs are thread-local, then objects allocated sequentially by a thread will likely be adjacent in memory, regardless of memory allocation activity by other threads. This ordering lends itself well to timelining and trying to determine the relationships between events.

4.4 Evaluation

Three case studies demonstrate RecOOPs ability to extract information from a Java runtime. In each case, only a memory image is available for analysis. Traditionally, understanding Java malware beyond sandboxing and behavioral analysis requires two things: the the JAR file and a decompilation tool such as CFR or JD-GUI. The analyst decompiles the JAR file, modifies the code, recompiles, and runs the code in an IDE such as Eclipse [45, 46, 47]. Obfuscation tricks can be very effective at blocking these efforts [48, 49, 50], and if the malware removes itself from the disk, extraordinary efforts are required to recover the original file, which may not be feasible. In this section, we show that Java processes contain copious amounts of information which lends itself to static forensic analysis.

4.4.1 Blackbox Malware Analysis and Reverse Engineering

We found an old version of the Adwind trojan on Malwr.org* and performed a Java centric analysis. We ran the malware on both Linux and Windows XP SP3 VMs and found that the malware appears to behave a little differently on Linux. Both versions of Java produce a similar thread listing (Table 4.5). However, the program behaviors diverge because the backdoor must dump a *native* library that is used by Java for snooping and keystroke logging.

Since this malware uses obfuscation, we explore the process for any latent buffers containing compressed data. Table 4.6 shows that files can either reveal information like passwords or contain unobfuscated class files. Listing 4.1 shows a high level prototype of a recovered class file created by Radare. The `extra/CLM.pass` reveals a password field, so enumerating the object and its field in the heap reveals the string value (`vooXN3UW`). Finding this value in a strings dump would be difficult.

4.4.2 Malware Proxy

To demonstrate the effectiveness of socket analysis, we wrote a program that simulates the basic capabilities of Java malware. In this case, an infection has been detected in the network, and an investigation of the system reveals malware acting as a network proxy. This proxy allows an external attacker to communicate with hosts on the internal network. Normally, the investigator may not be able to find out what information moved in and out of the network. However, Java's memory model allows the socket connections and buffered data to persist indefinitely.

We ran the simulation for five minutes, sending commands instructing the agents to "do something evil." Table 4.8 shows the recovered socket data. Since the buffered stream and subordinate objects were never collected or overwritten, most of the attackers commands remained intact. However,

*Windows analysis from Malwr.org: `https://malwr.com/analysis/NGUwYjM1OGI4MGE4NDZkYjg5ZGVhMGU4YTMyN2RlMDU/`

Thread Identifier	Native Address	Heap Address	Thread Name
1	0x00000000	0xa9e91020	main
1	0xb6907000	0xa4d3d050	main
2	0xb695f800	0xa4cd4e10	Reference Handler
2	0xb695f800	0xa9e90c58	Reference Handler
3	0xb6961000	0xa9e90ab0	Finalizer
3	0xb6961000	0xa4cd4c68	Finalizer
4	0xb697e000	0xa9e90938	Signal Dispatcher
4	0xb697e000	0xa4cd4af0	Signal Dispatcher
5	0xb697f800	0xa9e907c0	C1 CompilerThread0
5	0xb697f800	0xa4cd4978	C1 CompilerThread0
6	0xb6982c00	0xa4cd4800	Service Thread
6	0xb6982c00	0xa9e90648	Service Thread
7	0xa360ac00	0xa9e904a8	Java2D Disposer
7	0xa360ac00	0xa4cd4660	Java2D Disposer
8	0x00000000	0xa9e204f8	XToolkt-Shutdown-Thread
9	0xa361dc00	0xa4cd4318	AWT-XAWT
9	0xa361dc00	0xa9e90160	AWT-XAWT
10	0xa36f5800	0xa9e8fef8	Thread-0
10	0xa36f5800	0xa4cd15d8	Thread-0
11	0x09e24c00	0xa49f2720	Thread-1
13	0xb6907000	0xa49f9e58	DestroyJavaVM
14	0x09e35000	0xa49f5a78	pool-1-thread-1

Table 4.5 : Thread information extracted from the HotSpot JVM executing the Adwind malware on Linux.

Zip Inflater Address	Size	Decompressed Data
0xa48f46e0	181	\xca\xfe\xba\xbe ...
0xa48ff850	186	\xca\xfe\xba\xbe ...
0xa4901720	433	\xca\xfe\xba\xbe ...
0xa491fcd8	170	Manifest-Version: 1.
0xa4920228	170	Manifest-Version: 1.
0xa4924c20	15	plugins._008_
0xa492c590	170	Manifest-Version: 1.
0xa492cc10	477	\xca\xfe\xba\xbe ...
0xa9d87160	10	vooXN3UW
0xa9e88c50	819	\xca\xfe\xba\xbe ...
0xa9e89600	152	\xca\xfe\xba\xbe ...
0xa9e89f30	607	\xca\xfe\xba\xbe ...
0xa9e8a9c8	117	\xca\xfe\xba\xbe ...
0xa9e8ad70	727	\xca\xfe\xba\xbe ...
0xa9e8b700	1324	\xca\xfe\xba\xbe ...
0xa9e8c440	1008	\xca\xfe\xba\xbe ...
0xa9e8cf50	157	Manifest-Version: 1.
0xa9e8d2e8	157	Manifest-Version: 1.
0xa9e8d680	157	Manifest-Version: 1.
0xa9e8da58	157	Manifest-Version: 1.

Table 4.6 : Compressed class (*gray*), password (*red*), and other files found in the Java heap.

- offset -	0 1	2 3	4 5	6 7	8 9	A B	C D	E F	0123456789ABCDEF
0x00000000	cafe	babe	0000	0032	000d	0700	0b07	000c	.......2........
0x00000010	0100	0a61	6464	4172	6368	6976	6f01	0011	...addArchivo...
0x00000020	284c	6a61	7661	2f69	6f2f	4669	6c65	3b29	(Ljava/io/File;)
0x00000030	5601	000d	6361	7267	6172	506c	7567	696e	V...cargarPlugin
0x00000040	7301	0003	2829	5a01	000a	6765	7450	6c75	s...()Z...getPlu
0x00000050	6769	6e73	0100	1928	295b	4c70	6c75	6769	gins...()[Lplugi
0x00000060	6e73	2f41	6477	696e	6453	6572	7665	723b	ns/AdwindServer;
0x00000070	0100	0a53	6f75	7263	6546	696c	6501	0015	...SourceFile...
0x00000080	496e	7465	7266	6163	6550	6c75	6769	6e73	InterfacePlugins
0x00000090	2e6a	6176	6101	0018	706c	7567	696e	732f	.java...plugins/
0x000000a0	496e	7465	7266	6163	6550	6c75	6769	6e73	InterfacePlugins

0x00000000	cafe	babe	0000	0032	000a	0700	0807	0009	.......2........
0x00000010	0100	0e67	6574	496e	666f	726d	6163	696f	...getInformacio
0x00000020	6e01	0014	2829	4c6a	6176	612f	6c61	6e67	n...()Ljava/lang
0x00000030	2f53	7472	696e	673b	0100	0d67	6574	4d61	/String;...getMa
0x00000040	6341	6464	7265	7373	0100	0a53	6f75	7263	cAddress...Sourc
0x00000050	6546	696c	6501	0012	696e	7465	7266	6163	eFile...interfac
0x00000060	6549	6e66	6f2e	6a61	7661	0100	166f	7063	eInfo.java...opc
0x00000070	696f	6e65	732f	696e	7465	7266	6163	6549	iones/interfaceI
0x00000080	6e66	6f01	0010	6a61	7661	2f6c	616e	672f	nfo...java/lang/

Table 4.7 : Extracted class data for Adwind's plugin interface and survey functionality.

we found that some of the structural information of the messages was lost, because the proxy used `DataInputStream` to read the message length and command directly off the wire. While the message may not be intact, a forensic analyst could examine the class and method metadata and try to assemble a data flow graph, which could help recreate the message structure.

```
//  r2 -c 'java prototypes a' \
//   ae05bdff4d_1324_a9e8b700.class
import java.lang.ClassLoader;
import java.lang.String;
import java.util.zip.GZIPInputStream;
import extra.CLM;
import extra.CLM;
import extra.Constante;

class extra/CLM { // @0x0000
  // Fields defined in the class
  public static String pass;
  private static final extra.CLM instancia;

  // Methods defined in the class
  private void <init> ();
  public static extra.CLM getInstance ();
  public java.lang.Class findClass (String);
  private byte[] loadClassData (String);
  public static byte[] descomprime (byte[]);
  static void <clinit> ();
}
```

Listing 4.1: Radare2 `java prototypes` command reveals a custom loader in the decompressed class data.

4.4.3 Scripted Intrusion

We created a script that models how a *smash-and-grab* attacker would behave in a post-compromise setting. The scenario centers around an attacker exploiting the fact that dynamic plugins can be uploaded to a dotCMS server[†]. In this case the attacker leverages administrator credentials to upload and activate the plugin. When the plugin activates, it uses `wget` to retrieve and start the attacker's backdoor. The attacker uses a script to execute a series of steps using the malware. After each step in this script, we take a memory snapshot of the virtual machine and perform the JVM analysis. The implant relies on `ProcessBuilder` to execute system commands outside of the Java environment and has functions that allow the attacker to proxy and communicate with other systems, read

[†]`http://dotcms.com/`

Obj. Address	Remote Connection		In/Out	Data (Up to 30 Bytes)
0x91c779b8	10.18.120.18	48002	⇒	Do something evil-48002!
0x91c7ead0	10.18.120.18	48003	⇒	Do something evil-48003!
0x91c85b70	10.18.120.18	48002	⇐	s3cr3t_d4t3_48002-00000000s3cr
0x91c938d8	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDItMDAw
0x91c980d0	10.18.120.18	48003	⇐	s3cr3t_d4t3_48003-00000000s3cr
0x91ca5cb8	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDMtMDAw
0x91cbfef0	10.18.120.18	48004	⇒	Do something evil-48004!
0x91cc7008	10.18.120.18	48005	⇒	Do something evil-48005!
0x91ccdee8	10.18.120.18	48004	⇐	s3cr3t_d4t3_48004-00000000s3cr
0x91cdbad0	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDQtMDAw
0x91ce02c8	10.18.120.18	48005	⇐	s3cr3t_d4t3_48005-00000000s3cr
0x91cedeb0	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDUtMDAw

Table 4.8 : Recovered socket data (colored by the proxied connection) shows how the heap address forms a communication timeline.

and write files, download files, and interact with the OS.

This evaluation concentrates on the created processes, and how much information can gleaned from their Java artifacts. This script starts 73 processes that execute OS commands (e.g. `ls`, `ps`, etc.) Figure 4.2 shows how much command history is retained in the heap over time. Three garbage collection cycles are observable at $t = 25$, $t = 27$, and $t = 30$. Malware data exfiltration triggers the GC events.

The process objects accumulate and remain in memory even after several garbage collections.

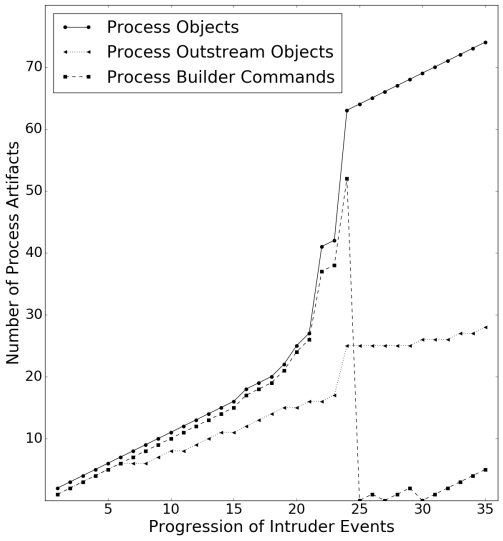


Figure 4.2 : Number of recoverable process artifacts present in the heap throughout the scripted attack.

Event Log from Extracted Process Builder and Process Information

Address	Process Builder Command	PID	Buffered Data
0x91cb7258	sudo cat /etc/sudoers	1242	#\n# This file MUST be edited w
0x91e1c6f8	uname -ar	1245	Linux java-workx32-00 3.19.0-1
0x91e2bd98	id -un	1247	java\n
0x91e3aff0	id -nG	1248	java adm cdrom sudo dip plugde
0x91e4a598	sudo cat /etc/passwd	1250	root:x:0:0:root:/root:/bin/bas
0x91eb1240	sudo cat /etc/shadow	1252	root:!:16678:0:99999:7:::\ndaem
0x91ec2da0	sudo netstat -ap		
0x91eefea8	ls -all	1273	total 10312\ndrwxrwxr-x 6 java
0x91f556f0	ls -all		
0x91f66498	sudo -S nmap [...]	1275	\nStarting Nmap 6.47 (http://n
0x91f7e400	sudo lsof		
0x91f8d810	mount -v	1298	sysfs on /sys type sysfs (rw,n
0x91ff7ce0	cat /root/.bash_history	1301	history — grep pg\n history — gr
0x92014d20	cat /home/java/.bash_history	1307	ifconfig\nsudo add-apt-reposito
0x9203ae78	ps -ef		
0x920623c0	zip -r /tmp/backup.dat.zip	1322	adding: home/java/.ssh/ (sto
0x920720a0	ps -ef		
0x92099530	ls -all /tmp/	1328	total 44\ndrwxrwxrwxrwt 9 root ro
0x920aab68	cat /tmp/backup.dat.zip	1333	PK\x03\x04\n\xdbL\x1fG\x0f\x1c
0x920cb718	ls -all /tmp/backup.dat.zip	1338	-rw-r–r– 1 root root 1617 Au
0x920db348	dd if=/dev/urandom of=		
0x920ead40	zip -r		
0x921601c8	ps -ef		
0x922c5dd8	ps -ef		
0x922ed348	zip -r /tmp/logs.zip /var/log/	1354	adding: var/log/ (stored 0%)
0x92305c50	ps -ef		

Table 4.9 : At $t = 21$, process artifacts are used to create an event log.

These processes also retain buffered data, which can be used to infer the commands executed on the system. Table 4.10 shows a sample of these buffered processes at the end of the experiment. We can see that an analyst could infer what 11 out of the 16 listed processes were doing. To verify this information, we refer back to $t = 24$ before the garbage collection wiped out most of the command history. Table 4.9 shows how the `ProcessBuilder` objects and the `Process` data buffers can be used to assemble an event log.

We also evaluate recovery of process artifacts from Volatility. For each memory image, the

Address	PID	Buffered Output
0x67020b20	1275	\nStarting Nmap 6.47 (http://n
0x67020c10	1403	total 176584\ndrwxrwxrwt 9 roo
0x6702de70	1273	total 10312\ndrwxrwxr-x 6 java
0x6702eb78	1252	root:!:16678:0:99999:7:::\ndaem
0x67092350	1250	root:x:0:0:root:/root:/bin/bas
0x67092b88	1248	java adm cdrom sudo dip plugde
0x670c0720	1245	Linux java-workx32-00 3.19.0-1
0x670c0880	1242	#\n# This file MUST be edited w
0x670c0f18	1354	adding: var/log/ (stored 0%)
0x670c13d8	1338	-rw-r–r– 1 root root 1617 Au
0x670c15d0	1333	PK\x03\x04\n\xdbL\x1fG\x0f\x1c
0x6711d068	1328	total 44\ndrwxrwxrwxrwt 9 root ro
0x6711d718	1322	adding: home/java/.ssh/ (sto
0x6714c8b0	1307	ifconfig\nsudo add-apt-reposito
0x6714cc10	1301	history | grep pg\nhistory | gr
0x6714ceb8	1298	sysfs on /sys type sysfs (rw,n

Table 4.10 : *Interesting* process output recovered at t=35.

Address	Calls	Method Name
0x63fdb6f8	256	Loader getLoaderInstance(...)
0x63fdb908	73	byte[] b64Decode(...)
0x63fdce98	256	integer sendSocketData(...)
0x63fdd038	256	integer sendSocketData(...)
0x63fdd670	1	void addClientHandlerSocket(...)
0x63fdd718	256	void stdout(...)
0x63fdd850	256	void logEvent(...)
0x63fddbd8	73	integer getPid(...)
0x63fddd50	73	integer startProcess(...)
0x63fddf68	256	java.lang.String readProcessStdout(...)
0x63fdd9e8	1	void main(...)
0x63fde1d8	1	integer access$100(...)
0x63fde168	2	java.lang.String access$000(...)
0x63fdb670	1	void start(...);

Table 4.11 : Call metadata for a selection of the `Loader`'s methods at t=35, revealing a large number of IO operations.

`linux_psview` and `linux_psxview` are used to try and find artifacts. Only one process (`sudo lsof`) could be accurately identified. This process runs from $t = 10$ through the end of the experiment. No other process artifacts could be recovered. We believe they were unrecoverable due to OS activity and memory volatility, because the relevant data structures were overwritten at some point after process termination and memory deallocation. This shows the limits Volatility and other similar frameworks, which do not currently account for runtime artifacts.

The HotSpot JVM produces telemetric data to help improve performance. `MethodCounters` are initialized on a method's first call, and it tracks the number of calls, which can be useful for malware analysis (Table 4.11). For example, if malware uses a `HashMap` to map specific commands to specific functions, understanding the malware's behavior is very challenging. The telemetric

information helps discern relevant functions from noise or potential obfuscation.

4.5 Future work

Future work should address several key challenges. First, our analysis tools can use a significant amount of memory while processing a malware memory image. Overhead results from creating environmental objects and values in addition to annotations which help support our analyses. The resulting memory consumption becomes an issue when memory image sizes are multiple gigabytes or when there are 100,000 or more individual objects.

Furthermore there is a semantic gap between some objects that prevents directly finding links between these objects without deeper analysis. At runtime, relevant information about an object can be determined with an API call. When the memory is analyzed in a static manner, those API calls are not available (even though the information that they use is typically in memory). This was the case with JAR file entry names and the compressed data from the entry in our experiment. A symbolic execution for VM bytecode (e.g. CLR, HotSpot, etc.) would help to eliminate this problem.

4.6 Conclusions

RecOOP is a memory analysis framework that helps generalize digital forensics of managed runtimes. We developed an implementation focused on the HotSpot JVM for Java 8. We also showed that the framework is practical for digital forensics and malware analysis, complementing other such tools.

Chapter 5

Future Work

Our work establishes a memory analysis tool chain for the HotSpot JVM, and it lays the groundwork for memory analysis in managed runtimes. This tool chain provides forensic and malware analysts with reverse engineering tools that work on multiple OS platforms. We also proposed a modular framework targeting the analysis of mobile malware and improving collaboration and analysis work flow. Our work improves the forensics fields, but there are several areas that require additional research.

5.1 Large Image Memory Analysis

System memory size is quickly exceeding the capacity of current tools. For example, the memory in a compromised server might exceed hundreds of GiB. Analyzing large memory images poses a number of different challenges. First, the memory size increases the search space for relevant artifacts and evidence. The increase in size also increases the intermediate data product along with the compute power required perform the final analysis. This increase in size demands more storage space for the initial memory image and the data byproducts.

Memory overlay systems take a binary blob and interpret it as a structure or value. The interpretation is based on a defined syntax that permits a robust definition of full or incomplete data structures. These representations can be challenging to implement because program memory layouts vary with OS, hardware architecture, and even the compiler used to create the program.

The primary challenge stems from the fact that overlays create several similar object types. When

overlays are applied to the binary data, excessive memory consumption results from several general cases. First, there is a *binary copy* and an *interpreted copy* of the value. Second, if the overlay interprets a com- plex value (e.g. an OOP data structure), then every interpreted OOP includes the type definition, values, and potentially a copy of the binary data. Finally, when binary data is lifted to runtime objects (e.g. binary blobs to Python instances) there is a penalty because the values in the overlay get expanded to the runtimes implementations of the overlay values. For programs with a small number of extracted artifacts, the penalty is negligible. However, as the number of artifacts rise above 5,000, the runtime performance deteriorates and the consumption of memory balloons.

A solution to this problem might be to implement a heap memory management system similar to the HotSpot JVM. The overlay system keeps a dictionary of known types, and each overlay object uses a structure that resembles an OOP. The overlay OOPs require at most four 8-byte words. The first word is the header or bit field specifying ownership and other relevant information. The second word points to the memory overlay information and code for any transformations. The third word points to the bytes interpreted by the overlay. The final word is an object or structure containing analysis annotations or other related information.

This approach alleviates the memory consumption because of redundant values and duplicate data structures. For the most part, the interpretation of values and structures is lazy, so the only time binary blobs are accessed is when they are needed. The general concept seems straight forward, but additional research and engineering are required to determine how to manage types and values when analyzing large memory images or images with many overlay objects.

Signatures or indicators of compromise (IOC) are used to identify malfeasance or threat actors operating in a network or on a host. Signatures come in many different forms. Initially, signatures started out as byte strings or regular expressions that were applied to network streams, memory images, and files. Anti-virus software relied heavily on these types of signatures.

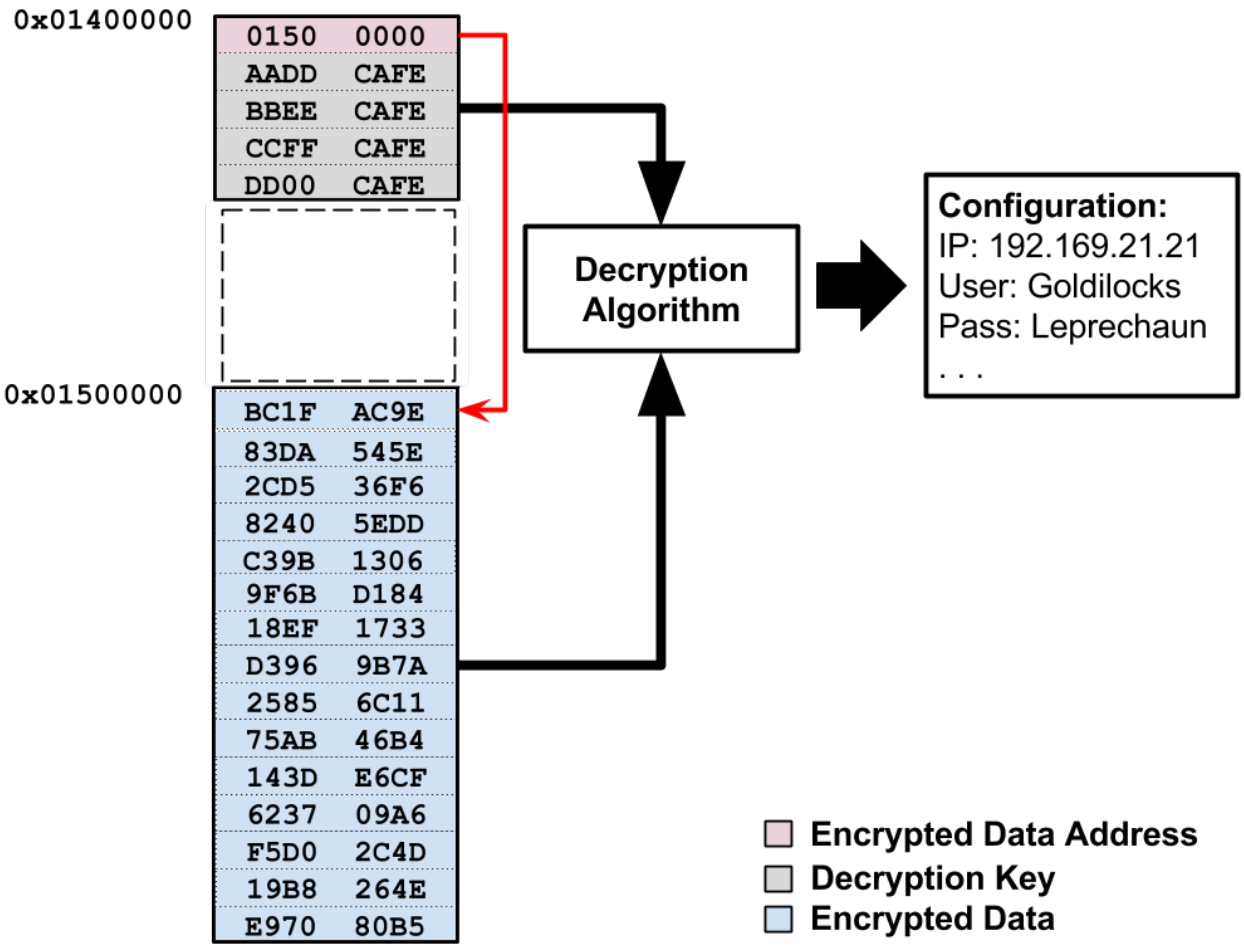


Figure 5.1 : This diagram shows some of the complexity involved in identifying and extracting signature matches in encrypted data.

Another type of IOC relies on program behaviors or machine state. Behavioral signatures use network, system, and process events to find anomalous behavior. Generally, systems that use behavioral signatures operate in an online manner, meaning the systems, network, etc. are running. However, these signatures can be applied on a system memory image along with events captured from the network. Other IOCs include file hashes, email content or addresses, observed URLs, malware, etc.

With all these different features, digital forensics and incident response have evolved into a data mining problem. Many different tools and technologies identify these indicators and report the

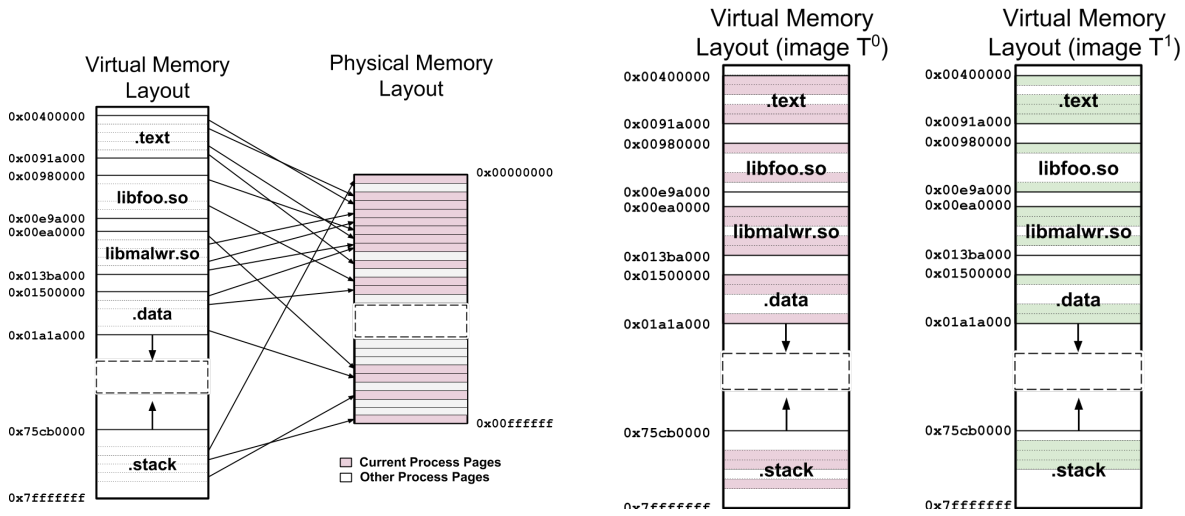
incident, and analysts respond in kind to mitigate the problem. However, memory forensics still lags because many of the tools and frameworks cannot incorporate this external state. Furthermore, memory analysis still relies mostly on signatures.

Unfortunately rule-based (e.g. chained signatures) tools are still very limited. For example, a Yara rule file can search for multiple strings and create conditional rules based on the presence of the string in memory. However, if the indicator is encoded, obfuscated, or encrypted in memory, identification or detection of these indicators fails. A stateful system with built-in decoders, disassemblers, and an understanding of well known data structures will help imrpove these systems. While the exact design is subject to future research, this analysis environment requires a grammar that helps define when to apply a rule, when to apply transformations, and how to handle the results of these analyses.

Future work in this area also needs to consider external events captured from monitoring services or servers. A framework integrating relevant network events into the memory analysis routine will help with rebuilding system level events. For example, captured historical events like registry writes or network communication can be combined with internal host connections to reveal lateral movement.

When these activities are detected, there is no defined approach or methodology for comparing system memory, let alone memory snapshots taken on the same system. Diffing memory will help reduce much of the analysis by comparing known good with bad, or by comparing memory at two different time periods, which can reveal important details.

Techniques for diffing memory images are necessary to improve automated memory analysis, but it has yet to be defined. Memory diffing can be used to compare a known good memory image to a known bad image. The process seems simple, but future research must devise a way to overcome the process virtual addressing, pages missing due to system activity, and finding ways to handle large memory images.



(a) Layout of the process in the system

(b) Comparing a process from two snapshots

Figure 5.2 : Diffing a process from the same host at two different times.

The research needs to determine the appropriate level of diffing. For example, performing a byte-for-byte comparison might yield poor results due to excessive noise. However, trying to diff memory at a data structure requires some internal knowledge about the programs components. Automatically learning these internals might be possible if enough samples are provided, because values that look like memory addresses can be used to build up potential structures. Finding the right balance is necessary since an analysts time can be constrained. Other attributes might also be considered for diffing memory. The amount used or current size of the process memory maps might prove useful to compare.

Comparing process memory between two different architectures is another interesting challenge. However, these architectures generally have architectural nuances preventing a direct comparison. Some of these nuances include word sizes, addressable memory boundaries (e.g. 4-byte boundary in RISC vs. 1-byte boundary for Intel x86), little endian versus big endian, etc. One approach might be to create an intermediate representation for values in memory and then attempt to diff that representation. This intermediate representation can also eliminate all the unnecessary data to

help make the comparison more effective.

Distributed memory analysis will become a necessity to handle very large memory images. To put this issue into context, host memory typically ranges between 4-16 GiB, but server memory can range between 32 GiB to multiple TiB. Memory forensics research presumes that a memory image will fit on a hard disk. However, as the size of memory increases, especially for servers, this presumption will be broken. Furthermore, large memory images require traversing the entire image to excavate relevant information. This analysis includes identifying malicious code, extracting console history, looking for injected libraries, etc. When the memory image is small, this approach is not a hindrance to the overall process of memory analysis.

Memory forensics will need to become more service oriented, handling different parts of memory separately. For example, when the memory is submitted for analysis the first step will be to enumerate the processes and kernel memory. Next, the memory will be carved up and submitted for detailed analysis. The analysis will extract artifacts like libraries, application code, services, etc. Objects, code, processes, and other artifacts are then classified to help expedite analysis and respond to the incident at hand.

5.2 Extend RecOOP to other architectures

We demonstrate that managed runtimes contain a plethora of information. When an attacker uses these runtimes for malicious activity, the managed memory can act as a history buffer. For example the HotSpot JVM retained the remnants from `stdout` from a number of terminated processes. This research can be extended to other runtimes like Microsoft .Net, Google V8 JavaScript Engine, and the 64-bit version of the HotSpot JVM.

Source code recovery from binary code structures is another important extension to our work. In some runtimes, program code is compiled before execution begins (e.g. AOT) or when code is

executed frequently (e.g. JIT). This compilation results in optimized native code, which helps the VM perform faster. When program bytecode is loaded, sometimes this code or a variant of it is retained as a safety mechanism.

Most code recovery efforts focus on lifting the *native code* into an intermediate language (IL), and then decompiling IL into a C or C++ like representation. However, this code could be reverted back into its deoptimized form (e.g. bytecode) and then decompiled. Each of these runtimes retain structures to revert to the original code because the optimized code could be an incorrect version of the original function.

One reason for addressing this challenge is due to attackers creating functionality in memory without a *tangible* program. Specifically, Microsoft Powershell gives attackers and administrators the power to submit C# code for a task. The C# code is compiled by a VM and then added to the environment. The underlying data structures that represent the code in the VM can be used to recover the source submitted by the attacker.

Multi-architecture emulation or symbolic execution should also be addressed. There are several frameworks that support symbolic execution and emulation for forensic purposes. In general, these frameworks typically only support one architecture at a time. However, managed runtimes generally support executing *native code* alongside the managed code.

This presents a challenge in several respects. First, frameworks exclude managed runtime bytecode execution support. They tend to focus only on native architectures like ARM, x86, x86-64, and MIPS. Second, when frameworks begin to support these *dual* environments, they must be aware of the VM, VM contexts (e.g. native or managed), understand how values are marshaled between environment, etc. Finally, the resulting framework should be able to discern between native code, AOT or JIT native code, and VM bytecodes.

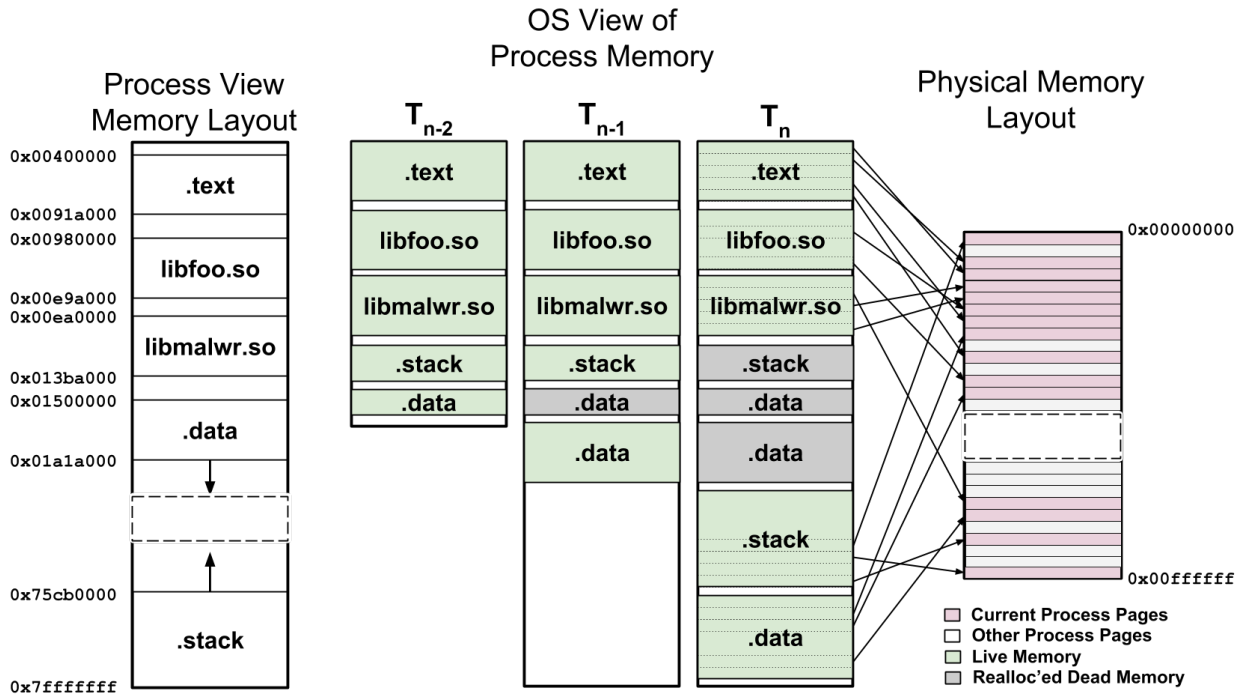


Figure 5.3 : Using similar memory allocation methods from generational GC runtimes could improve malware analysis. As time passes, new and old memory allocations for segments of memory can be compared to identify program behavior.

5.3 Improving Malware Analysis with Managed Memory

Virtualization enables the construction of economical malware sandboxes. These sandboxes are virtual machines (e.g. virtual hardware and OS) with well-known applications that attackers attempt to exploit. These sandboxes typically monitor memory reads and writes, look for injected libraries or machine code, capture traffic, watch for several other system modifications.

Design and development of a sandbox is challenging for several reasons. First, low-level instrumentation is required to capture call sites, read call parameters, etc. Second, each patch could bring a wave of new changes to the modified OS, so the sandbox environment may be hard to maintain. Third, creating and maintaining a realistic environment for the malware to run in requires a great deal of effort. Finally, performing analysis on the resulting memory reads and writes

produces a significant amount of data. All this data must be thoroughly analyzed to extract meaningful information. For common malware, this process is easy, but state-sponsored malware or very sophisticated attackers can evade sandboxes easily.

One way around the cost of maintaining an instrumented sandbox is to focus purely on black box analysis. For example, instead of instrumenting the VM, just let it run without any intervention, and simply record the state at the end of the execution. Unfortunately, this approach can miss vital information.

An alternative is to use a managed memory system. In this case the managed memory is a large pool of memory that works like a Java heap. When a program makes an allocation, the OS or hypervisor fulfills the allocation request and passes a pointer back to the process. This approach is attractive because memory allocations can be monitored, and as time marches forward, there is an understanding about when reads and writes take place. This approach could also make memory diffing much easier.

In theory this sounds simple, but there are a few challenges ahead. First, allocating memory in this manner might lead to out of memory segmentation faults, so the process or hypervisor might need to perform GC. In this case, the GC is simply compaction, where *live memory* is simply moved closer to the start of the heap. At each GC, the memory regions can be labeled and dumped for analysis and diffing over time. However, this GC process leads to a second challenge.

When the memory addresses are moved, all the raw pointers in the heap are invalidated. This means the raw-pointer access needs to be regulated with an address translation mechanism. Also, if this sandboxing technique is implemented in the OS; the OS will be limited to controlling a set of processes memory allocation. If the malware injects into another process the OS may not be able to maintain the memory tracking or address translation mechanism.

This means the most effective place to implement and control this type of allocation is in the

hypervisor. In this case, the hypervisor could attempt to use linear addressing and allocation for all programs and processes. Otherwise, the hypervisor could simply be informed of a process to track and work with the OS on providing memory to the process on each allocation. If malware injects into the process, the hypervisor can be used to mark the victim process, and then that processs memory allocation can be switched to the managed memory model.

Chapter 6

Conclusions

The HotSpot JVM is one of the most prominent managed runtimes in enterprise networks. Attackers recognize that they can write malware that runs on multiple types of hardware architectures and operating systems. Writing malware in these managed runtimes also helps them evade most endpoint security technologies that are not able to perform program inspection.

Furthermore, attackers and threat actors only need to succeed once to be successful. Many organizations are not prepared to deal with the aftermath of a successful attack. The response requires understanding the most current state of the network, which requires a mix of network traffic analysis, logs from a myriad of sources, and most importantly memory dumps from the affected systems.

Understanding and extracting relevant evidence from memory dumps is very challenging, and the difficulty only increases when there are no plugins or tools to support specific processes or programs. Due to the prevalence and other managed runtimes, we have focused our efforts on understanding and recovering information from HotSpot processes. First, we showed that managed runtimes have the potential to retain artifacts for a significant period. Then we developed a methodology for tackling the memory analysis of managed runtimes.

We exploit the fact that the internal memory management system performs very little, if any data sanitization. Managed runtimes also use internal data structure information that contains all the loaded type information. We show that these structures can be mined in a consistent manner to recover any objects that were present in the heap. This information can be used to rebuild the data type and member fields without any access to the source code. Furthermore, the recovered byte

code can be decompiled into the original language, and this can be used for analysis.

Even though system memory recycles overtime, we found that some of the history, which may be absent from the operating system data structures, may still be recoverable. For example, processes started from Java remained in the heap for extended periods of time. These commands were only overwritten when the simulated attacker exfiltrated large amounts of data.

Memory analysis is slowly moving off the OS stack. This stack builds on hardware abstraction layers and the operating system, but now forensic analysts need to be able to analyze prevalent applications, especially when these applications are companys production and revenue streams. In a modern information technology infrastructure, the cost of tearing down and rebuilding a virtual machine is cheap. Capturing the memory is also inexpensive, but analyzing this memory to determine how a Java server was compromised is expensive. We have developed a methodology and tool set that helps reduce this cost.

Bibliography

[1] R. Langner, "To kill a centrifuge: A technical analysis of what stuxnets creators tried to achieve," 2013.

[2] E. Galperin, C. Quintin, M. Marquis-Boire, and C. Guarnieri, "I Got a Letter From the Government the Other Day...:Unveiling a Campaign of Intimidation, Kidnapping, and Malware in Kazakhstan," 2016.

[3] V. Kamluk, C. Raiu, and I. Soumenkov, "THE ICEFOG APT: A TALE OF CLOAK AND THREE DAGGERS," September 2013.

[4] V. Kamluk, C. Raiu, and I. Soumenkov, "The icefog apt hits us targets with java backdoor," January 2014.

[5] J. Scott-Railton, M. Marquis-Boire, C. Guarnieri, and M. Marschalek, "Packrat: Seven years of a south american threat actor," December 2015.

[6] A. Drozhzhin, "Adwind malware-as-a-service hits more than 400,000 users globally," February 2016.

[7] R. W. Smith and A. Pridgen, "STAAF: Scaling android application analysis with a modular framework," in *System Science (HICSS), 2012 45th Hawaii International Conference on*, IEEE, 2012.

[8] A. Pridgen, "Reversing java (malware) with radare."

[9] R. E. Jones and C. Ryder, "A study of java object demographics," in *Proceedings of the 7th*

international symposium on Memory management, ACM, 2008.

[10] Sun Microsystems, "Memory management in the Java HotSpot virtual machine," Apr. 2006.

[11] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th international symposium on Memory management*, ACM, 2004.

[12] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, 2004.

[13] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, 2005.

[14] J. Viega, "Protecting sensitive data in memory," 2001.

[15] V. DSilva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *IEEE CS Security and Privacy Workshop*, (San Jose, CA), 2015.

[16] K. Gondi, A. P. Sistla, and V. Venkatakrishnan, "Deics: Data erasure in concurrent software," in *Secure IT Systems*, Springer, 2014.

[17] M. Anikeev and F. Freiling, "Preventing malicious data harvesting from deallocated memory areas," in *Proceedings of the 6th International Conference on Security of Information and Networks*, ACM, 2013.

[18] M. Anikeev, F. C. Freiling, J. Götzfried, and T. Müller, "Secure garbage collection: Preventing malicious data harvesting from deallocated java objects inside the Dalvik VM," *Journal of Information Security and Applications*, vol. 22, 2015.

[19] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: limiting

mobile data exposure with idle eviction," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[20] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, osdi vancouver," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[21] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, IEEE, 2007.

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, May 2009.

[23] A. Case, "Memory analysis of the Dalvik (Android) virtual machine," Dec. 2011.

[24] T. Müller and M. Spreitzenbarth, "FROST: Forensic recovery of scrambled telephones," in *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, 2013.

[25] C. Hilgers, H. Macht, T. Müller, and M. Spreitzenbarth, "Post-mortem memory analysis of cold-booted Android devices," in *Proceedings of the 2014 Eighth International Conference on IT Security Incident Management & IT Forensics*, IMF '14, 2014.

[26] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, "Recovering c++ objects from binaries using inter-procedural data-flow analysis," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, ACM, 2014.

[27] J. Stttgen and M. Cohen, "Robust linux memory acquisition with minimal target impact," *Digital Investigation*, vol. 11, Supplement 1, 2014. Proceedings of the First Annual DFRWS Europe.

[28] M. Cohen, "Rekall memory forensics framework," in *DFIR Prague 2014*, SANS DFIR, 2014.

[29] S. VöMel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the Windows operating system," *Digit. Investig.*, vol. 8, July 2011.

[30] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," *ACM Sigplan Notices*, vol. 41, no. 10, 2006.

[31] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Io-lite: A unified i/o buffering and caching system," in *Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, Feb. 1999.

[32] A. Pridgen, S. Garfinkel, and D. S. Wallach, "Present but unreachable: Reducing persistent latent secrets in HotSpot JVM," in *System Science (HICSS), 2017 50th Hawaii International Conference on*, IEEE, 2017.

[33] A. Walters, "The Volatility Framework: Volatile memory artifact extraction utility framework," 2007.

[34] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "DSCRETE: Automatic rendering of forensic information from memory images via application logic reuse," in *Proceedings of the 23rd USENIX conference on Security Symposium*, USENIX Association, 2014.

[35] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing together android app guis from memory images," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[36] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "VCR: App-agnostic recovery of photographic evidence from Android device memory images," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[37] Y. Li, "Where in your ram is "*python san_diego.py*"?," 2015.

[38] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," 2010.

[39] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution.," in *NDSS*, vol. 8, 2008.

[40] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009.

[41] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures.," in *NDSS*, 2011.

[42] G. G. Richard III and V. Roussev, "Scalpel: A frugal, high performance file carver.," in *DFRWS*, 2005.

[43] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 21–29, May 2007.

[44] pancake, "Radare2: a unix-like reverse engineering framework and commandline tools," 2015.

[45] K. Chen and J.-B. Chen, "On instrumenting obfuscated Java bytecode with aspects," in *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, SESS '06, 2006.

[46] T. A. Proebsting and S. A. Watterson, "Krakatoa: Decompilation in Java (Does bytecode reveal source?).," in *COOTS*, 1997.

[47] S. Cimato, A. De Santis, and U. F. Petrillo, "Overcoming the obfuscation of Java programs by identifier renaming," *Journal of systems and software*, vol. 78, no. 1, 2005.

[48] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for Java bytecode," *Journal of Systems and Software*, vol. 71, no. 1, 2004.

[49] D. Low, "Protecting Java code via code obfuscation," *Crossroads*, vol. 4, no. 3, 1998.

[50] J. Schlumberger, C. Kruegel, and G. Vigna, "Jarhead analysis and detection of malicious Java applets," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, 2012.